VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

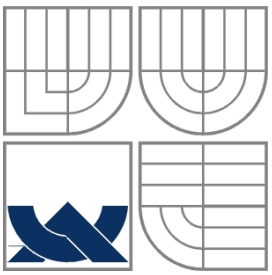
TRANSFORMACE JAZYKA C DO VHDL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

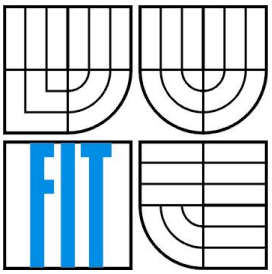
AUTOR PRÁCE
AUTHOR

Bc. Martin Mecera

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

TRANSFORMACE JAZYKA C DO VHDL

TRANSFORMATION FROM C TO VHDL LANGUAGE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. Martin Mecera

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. Karel Masařík Ph.D.

BRNO 2010

Abstrakt

Práce popisuje proces transformace chování procesoru popsaného v jazyku C do jazyku VHDL. Jednotlivé kroky automatizované transformace jsou porovnány oproti manuálnímu návrhu procesoru. Práce vyzdvihuje výhody vnitřní reprezentace programu ve formě grafu. V práci jsou uvedeny optimalizace založené na několika faktorech. Jedním z nich jsou algebraické úpravy výrazů. Vhodnou aplikací vlastností matematických operátorů – asociativity, komutativity a distributivity – lze snížit dobu výpočtu nebo omezit prostorovou náročnost výpočtu. Zvláštní pozornost je věnována optimalizacím, které využívají paralelizmus dílčích výpočetních operací k plánování. Jsou diskutovány algoritmy plánování omezeného časem a prostorem. Práci uzavírá kapitola o alokaci zdrojů.

Abstract

The thesis describes the process of transformation of the behavior of processor described in C language into VHDL language. Individual steps of automatized transformation are compared to manual design of processor. The thesis highlights advantages of the internal representation of program in the form of graph. Optimizations based on various factors are introduced in this thesis. One of them are algebraic modifications of expressions. The time of computation or space requirements of the circuit can be lowered by proper aplication of properties of math operators – associativity, comutativity and distributivity. Special attention is payed to optimizations, that make use of parallelism of operations for the process of operation scheduling. Algorithms of time-constrained scheduling and resource-constrained scheduling are discussed. The end of this thesis is devoted to resource allocation.

Klíčová slova

jazyk VHDL, jazyk C, transformace, jazyk ISAC, paralelizmus, procesor, graf, optimalizace, plánování, alokace

Keywords

VHDL language, C language, transformation, ISAC language, parallelism, processor, graph, optimization, allocation

Citace

Martin Mecera: Transformace jazyka C do VHDL, diplomová práce, Brno, FIT VUT v Brně, 2010

Transformace jazyka C do VHDL

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Karla Masaříka, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Bc. Martin Mecera
27. dubna 2010

Poděkování

Děkuji vedoucímu práce Ing. Karlu Masaříkovi, Ph.D., za odbornou pomoc.

Obsah

1 Úvod

Mikroprocesory lze navrhovat v různých jazycích. Nejčastěji je návrh realizován v jazycích pro popis hardwaru (např. VHDL nebo Verilog). Nevýhodami jazyků pro popis hardwaru jsou zejména obtížné odhalování chyb, nízká úroveň popisu, vysoké nároky na znalosti a zkušenosti návrháře. Další skupinou jazyků pro návrh mikroprocesorů jsou smíšené jazyky pro popis architektury. Mezi tyto jazyky patří jazyk ISAC (*Instruction Set Architecture C*)[19]. Jazyk ISAC byl vytvořen na Fakultě informačních technologií Vysokého učení technického v Brně.

Jazyk ISAC kombinuje deklarativní popis struktury mikroprocesoru s procedurálním popisem chování mikroprocesoru (v jazyku C). Z popisu v jazyku ISAC je vygenerován popis v jazyku VHDL a rychlý simulátor ve vyšším programovacím jazyku (např. C). Simulátor koresponduje s modelem mikroprocesoru v jazyku VHDL. Díky této korespondenci lze efektivně odhalovat chyby v návrhu mikroprocesoru. Výhodou jazyka C je to, že odstiňuje návrháře od nízkoúrovňového popisu hardwaru. Vyšší nároky jsou kladeny na algoritmus transformace místo na znalosti návrháře. Tato práce se zabývá generováním modelu mikroprocesoru v jazyku VHDL z popisu chování mikroprocesoru v jazyku C resp. z jazyku ISAC.

Druhá kapitola pojednává o obecných rysech procesorů. Autor srovnává skupiny procesorů a uvádí schéma obecného procesoru.

Třetí kapitola je věnována modelu procesoru v jazyku ISAC. Kapitola rozebírá stavební bloky popisu procesoru v jazyku ISAC. Autor srovnává klasický postup návrhu mikroprocesoru s návrhem v jazyku ISAC a následnou automatizovanou transformací do jazyku pro popis hardwaru.

Čtvrtá kapitola zahajuje popis jednotlivých kroků automatizované transformace. Ve čtvrté kapitole je pojednáno o vnitřním popisu modelu chování – abstraktním syntaktickém stromu.

Vnitřní reprezentace výpočtu algoritmu bývá obvykle zaznamenávána ve formě grafu. V páté kapitole je popsán proces konstrukce grafu řízení výpočtu. Graf řízení výpočtu podává obraz o větvení výpočtu.

Šestá kapitola se věnuje konstrukci grafu datových závislostí. Graf datových závislostí je vstupem algoritmů optimalizace.

Kapitoly sedm, osm a devět pojednávají o optimalizaci vyhodnocování matematických výrazů. Optimalizace jsou založeny na algebraických úpravách. V sedmé kapitole je představen Huffmanův algoritmus pro redukci výšky stromu výrazů. Algoritmus vychází ze známého Huffmanova kódování.

V kapitole osm je ukázáno, že optimalizovat výpočet lze i pro asociativní operátory, pokud doba výpočtu je při různých uzávorkováních odlišná. Jako příklad takového operátoru bylo zvoleno násobení matic.

Devátá kapitola seznamuje čtenáře s optimalizací založenou na distributivních operátorech. Vhodnou aplikací distributivního zákona lze ušetřit výpočetní jednotky. Je ukázáno, že algoritmy hledající optimální řešení jsou nutně těžké. Jeden z takových algoritmů je v textu navržen a analyzován z pohledu časové složitosti.

Významná část práce se věnuje plánování operací. Mezi důležité směry plánování patří plánování omezené časem a plánování omezené prostorem. Algoritmy optimálního plánování bývají obecně těžké. Patří mezi ně celočíselné lineární programování. Jeho výhoda spočívá v tom, že bývá snadnější navrhnout řešení problému pomocí lineárních rovnic a nerovnic než psaní specializovaného algoritmu. Výzkum v oblasti plánování začíná často tím, že je optimalizační úloha specifikována tímto způsobem. V rámci kapitoly plánování je celočíselnému lineárnímu programování věnován velký prostor. Autor této práce navrhl vlastní metodu plánování omezeného prostorem, která je právě založena na celočíselném lineárním programování. Ostatní podkapitoly plánování se věnují heuristickým algoritmům, jejichž časová složitost bývá polynomiální.

V kapitole 11 o alokaci zdrojů je pojednáno o mapování paměťových a výpočetních částí algoritmů na fyzické zdroje systému. Algoritmy alokace se snaží o minimalizaci ceny obvodu. Podobně jako u plánování se jedná o obecně těžké problémy. Uvedené algoritmy jsou proto založené na heuristikách.

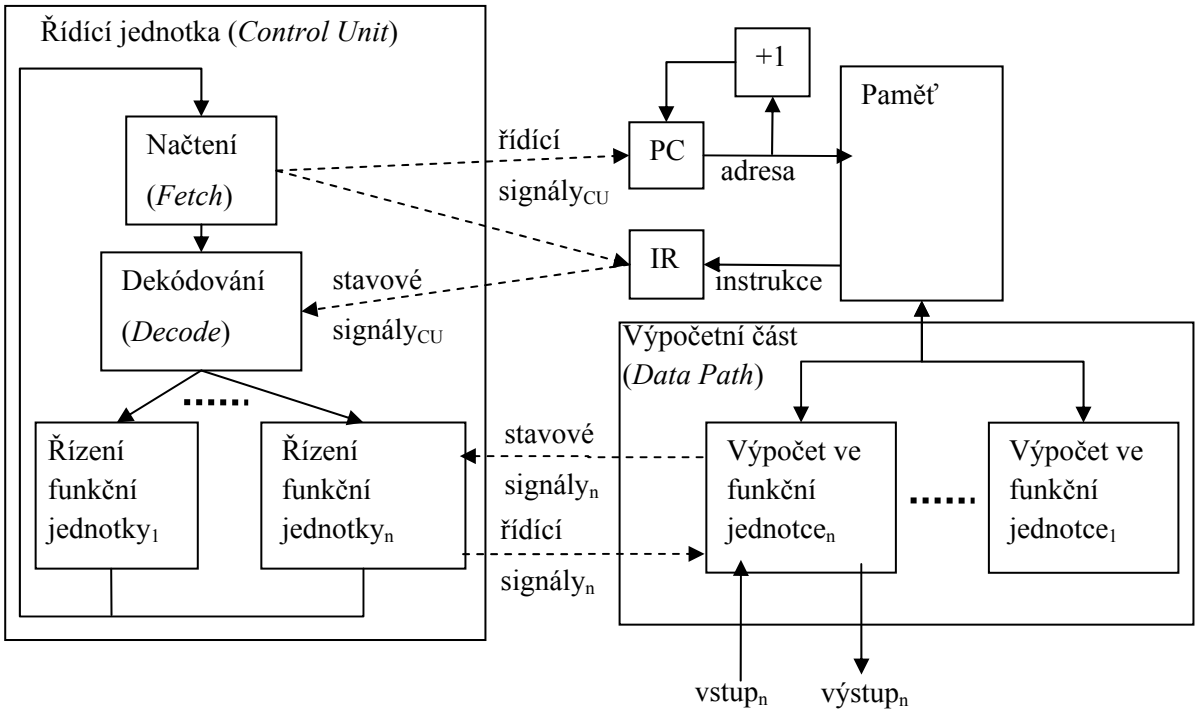
V závěru této práce jsou shrnuty autorovy přínosy a návrhy k dalšímu rozšíření.

2 Procesor

Procesory lze rozdělit do dvou skupin: jednoúčelové (dedikované) a pro obecné použití (*general purpose processor*). Dedikovaným se také říká aplikačně specifické (ASIC). Funkce vykonávaná dedikovaným procesorem je do procesoru napevno zakódována. Procesor obecného použití se od dedikovaného liší tím, že čte instrukce z paměti, provádí dekódovaní a iniciuje spuštění výpočtu ve specializované funkční jednotce. Funkčních jednotek je v procesoru několik druhů. Například rodina procesorů P6 firmy Intel© obsahuje jednotky: load/store, výpočty v pevné řádové čárce (*Integer Execution Unit*), skoky (*Jump Execution Unit*), multimediální vektorové výpočty MMX (*MMX Execution Unit*), výpočty v plovoucí řádové čárce (*Floating-Point Execution Unit*) [1]. Některé funkční jednotky mohou být v procesoru obsaženy vícenásobně. Tím je umožněno vykonávat stejné typy častých instrukcí paralelně. V procesorech P6 jsou zdvojeny jednotky výpočtu v pevné řádové čárce a MMX.

V moderních procesorech existují pokročilé techniky paralelního vykonávaní instrukcí. Procesory obvykle obsahují jednotku *Reorder Buffer* (ROB) a rezervační stanici. Rezervační stanice umožňuje odložit zpracování instrukce, která čeká na operandy. Zpracování pak probíhá způsobem data-flow [2]. Diplomová práce se touto problematikou nezabývá.

Na funkční jednotky lze pohlížet jako na dedikované procesory, protože také provádějí předem určenou operaci. Pohled na obecný procesor je na obrázku 1.



Obrázek 1: Schéma procesoru pro obecné použití. Převzato z [3] a upraveno. Řídící a stavové signály jsou mezi každou jednotkou řízení a její výpočetní jednotkou.

3 Model procesoru v jazyku ISAC

ISAC je smíšený jazyk pro popis architektury. Kombinuje deklarativní popis s procedurálním. Deklarativně jsou popsány zdroje (např. registry, paměti, externí logické obvody) a výpočetní celky (operace). Procedurálně (v syntetizovatelné podmnožině jazyka C) je popsán výpočet v operacích. Jazyk ISAC se částečně podobá jazyku LISA (*Language for Instruction Set Architecture*)[5]. V jazyku ISAC lze popsat systém na čipu.

Popis systému v jazyku ISAC se skládá částí

- **`RESOURCE`** – deklarace zdrojových prvků,
- **`OPERATION`** – popis výpočtu, tzv. operace,
- ostatních částí mimo **`RESOURCE`** a **`OPERATION`**.

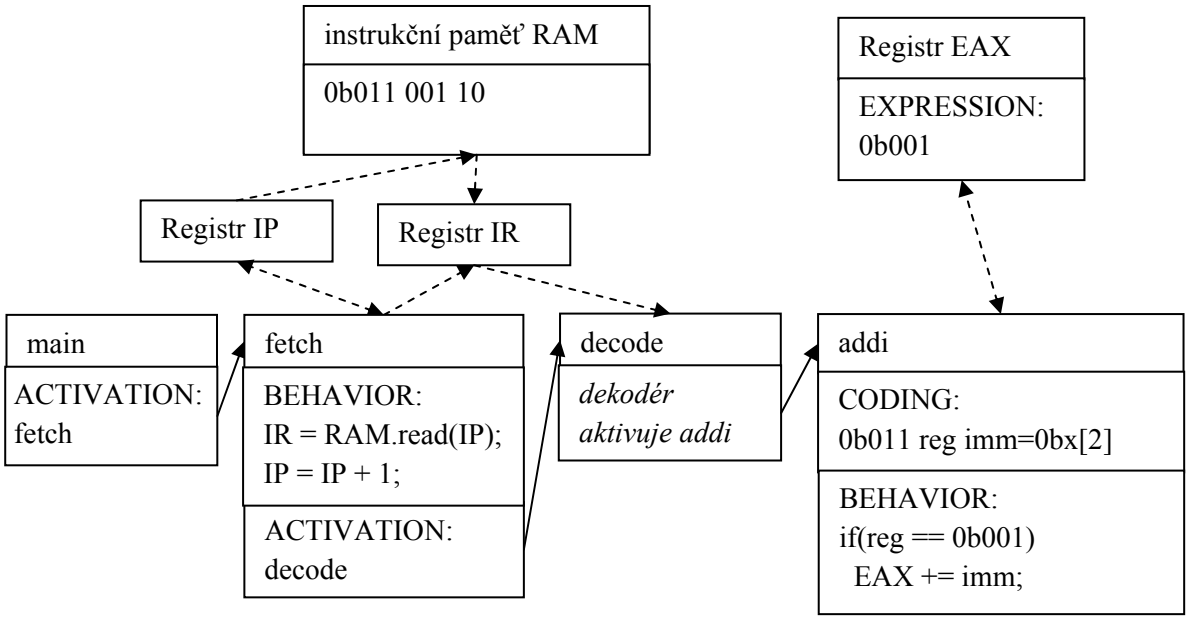
Zvláštní operací je operace *main*. *Main* je spouštěcí bod systému. Podobně existuje funkce *main* v jazyku C. Rozdíl mezi jazykem ISAC a C je v tom, že po skončení funkce *main* v C je program ukončen, ale v jazyku ISAC se běh vrací na začátek operace *main*. Operace *main* je zodpovědná za spouštění (aktivaci) ostatních operací. Obvykle bývá jako první aktivována operace fetch pro načtení instrukce do instrukčního registru.

Ukončení vykonání jedné operace může způsobit zahájení závislé operace. Např. po dokončení operace fetch může následovat operace decode. V jazyku ISAC se následná aktivace operace popisuje v sekci **`ACTIVATION`**. Podmíněné spuštění chování lze realizovat v sekci **`BEHAVIOR`**. Například lze spustit chování operace X, pokud je v registru eax hodnota 0b01.

Sekce **`BEHAVIOR`** se nachází uvnitř sekce **`OPERATION`**. V sekci **`BEHAVIOR`** se nachází popis chování dané operace. Chování je popsáno v syntetizovatelné podmnožině jazyka C. Spuštění chování operace v jazyku C je stejné jako volání funkce bez parametrů. Pokud například existuje operace alu, pak její spuštění v C odpovídá volání stejnojmenné funkce bez parametrů: „ *alu();* “.

Další způsob aktivace operace nabízí instrukční dekodér. Instr. dekodér je obvod, který čte instrukce v jazyce symbolických instrukcí a podle prefixu instrukce aktivuje odpovídající operaci.

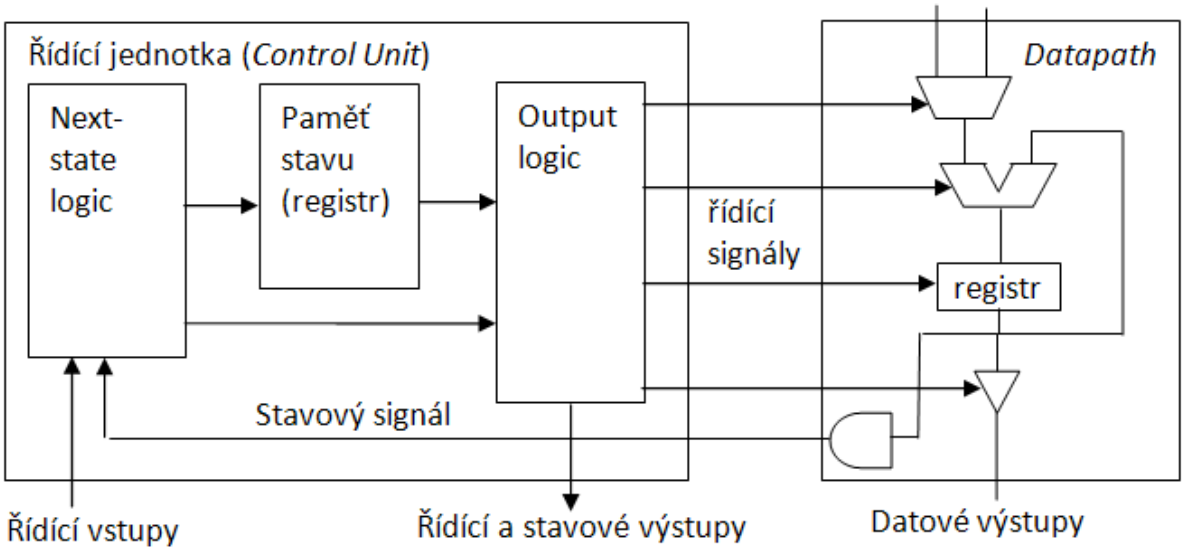
Obrázek 2 znázorňuje aktivaci operací vyvolanou sekcí **`ACTIVATION`** a aktivaci vyvolanou dekodérem instrukcí. Operace *addi* v sekci **`CODING`** říká, že má prefix 0b011 a argumenty jsou: číslo registru a dvoubitová hodnota. Registr **`EAX`** v sekci **`EXPRESSION`** říká, že číslo registru je 0b001. Operace *fetch* načte do instrukčního registru (**`IR`**) instrukci na adrese odkazované registrem **`IP`** (*instruction pointer*). Operace *fetch* poté inkrementuje **`IP`**. Po op. *fetch* je aktivován instrukční dekodér. Dekodér načte instrukci 0b011 001 010 z instrukčního registru. Podle prefixu 0b011 zjistí, že má aktivovat operaci *addi*. Po aktivaci operace *addi* dojde k vykonání sekce **`BEHAVIOR`**. V sekci **`BEHAVIOR`** je nejprve zjištěno číslo registru a následně je k danému registru přičtena hodnota. V našem příkladu se k **`EAX`** přičte hodnota 2 (0b10).



Obrázek 2: Ukázka aktivace operací.

3.1 Operace

Funkčním jednotkám budeme dále v textu říkat operace a rozlišíme jejich řídící a výpočetní část. Obvod pro vykonávání operací se skládá ze dvou částí – řídící jednotky (*Control Unit*) a Datapath. Řídící jednotka je konečný automat (FSM – *Finite State Machine*), jehož stavy odpovídají částem výpočtu v jednom taktu. Datapath je obvod v němž se samotný výpočet provádí. Výpočet v datapath je rozdělen na úseky, které se vykonávají po taktech. Data jsou mezi výpočty uchovány v registrech. Výpočet nad daty je prováděn v kombinačních obvodech, které se nacházejí mezi registry. Řídící jednotka spouští kroky výpočtu v datapath. Vodiče kterými je spouštěn výpočet se nazývají řídící. Vektor hodnot řídících vodičů se nazývá kontrolní slovo (*control word*). Datapath informuje řídící jednotku o stavu výpočtu pomocí stavových signálů. Hodnoty do stavových signálů bývají vysílány komparátory nebo logickými členy. Řídící jednotka bývá informováno o porovnání hodnot v registrech a v závislosti na výsledku řídí, kterou větví se bude výpočet ubírat. Řídící jednotka v procesoru obecného použití komunikuje s okolím pomocí řídících signálů. Patří mezi ně reset, finished a signály ovládající sdílené prostředky. Reset uvádí CU do iniciálního stavu. Finished informuje nadřazenou entitu o dokončení operace. Typickým sdíleným prostředkem je paměť, pro kterou jsou generovány řídící signály Enable, Read request, Write request. (Pojmenování se může lišit podle výrobce a typu paměti). Datapath komunikuje s okolím prostřednictvím datových signálů. Typicky čte a zapisuje data z/do sdílených registrů a pamětí, počítá adresu v paměti pro čtení operandů nebo zápis výsledku. Schéma funkční jednotky je na obrázku 3.



Obrázek 3: Schéma funkční jednotky. Převzato z [3] a upraveno.

3.2 Obvod funkční jednotky

Obvod funkční jednotky lze sestavit manuálně nebo automatizovaně transformací z vyššího programovacího jazyka.

Manuální proces zahrnuje:

1. Rozdělení algoritmu na výpočetní bloky, které lze vykonat v jednom taktu.
2. Pokud je součástí algoritmu atomická operace s latencí vyšší než jeden takt, pak má autor na výběr z možností:
   2.1. Zahájit operaci v dřívějším taktu a vykonávat jiné operace, které spolu nekolidují.
   2.2. Zahájit operaci a čekat potřebný počet taktů na její dokončení. Během čekání neprovádět jiné výpočty.
3. Plánování vykonání resp. zahájení a ukončení výpočtů do příslušných taktů. Výpočet může být vykonán dřív než vyžaduje algoritmus (tj. paralelně s jiným výpočtem), pokud nebudou porušeny datové závislosti.
4. Vytvoření konečného automatu, jehož stavy odpovídají taktům.
5. Sestavení výpočetního obvodu (datapath). Datapath je rozdělen na úseky výpočtu, které lze vykonat v jednom taktu. Předávání dat mezi těmito úseky probíhá přes paměťové prvky obvodu (např. registry).
6. Převedení konečného automatu na řídící jednotku – definuje řídící a stavové signály, sestaví obvod následujícího stavu (*next-state logic*) a obvod výstupu (*output logic*).
7. Napojení řídících a stavových signálů mezi řídící jednotkou a datapath.
8. Definice rozhraní pro komunikaci funkční jednotky s okolím. Jedná se o řídící a datové vstupy a výstupy.

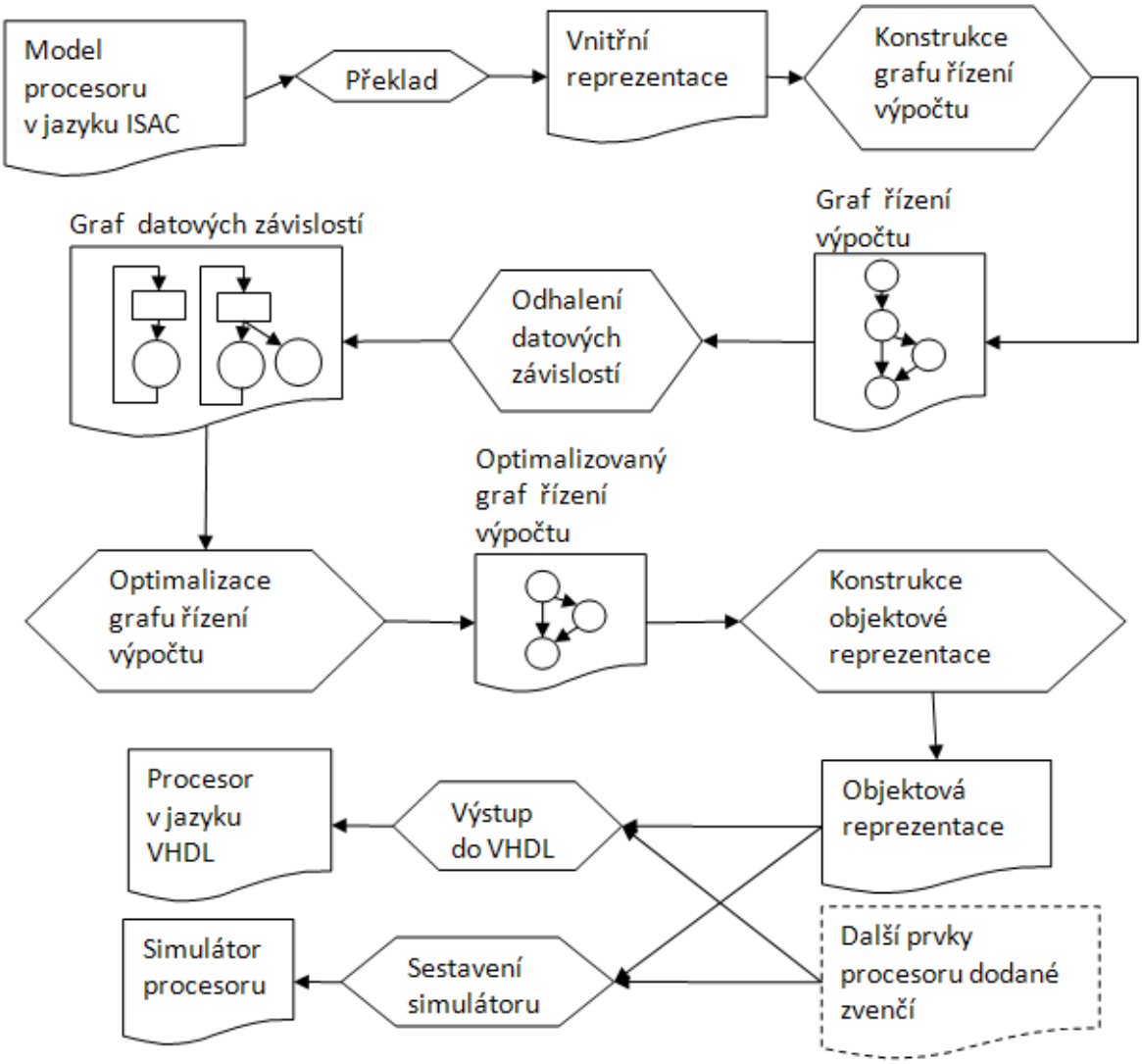
Komplikované mohou být především body 2 a 3. Bývá obvyklé, že zdrojů na čipu je omezené množství. Vývojář by ideálně měl naplánovat výpočty tak, aby celkový algoritmus proběhl v co nejkratším čase a zároveň aby prostředků využil co nejméně. Čím méně prostředků využije, tím levnější čip bude. Tento problém je z kategorie NP-úplných problémů. Optimální řešení lze nalézt v

rozumném čase pro jednoduché algoritmy. Pro komplikovanější algoritmy, které dovolují více přeuspořádávat výpočty, existují heuristické postupy nalezení suboptimálního řešení. Mezi algoritmy hledající optimální řešení patří *Integer Linear Programming*, mezi heuristické např. *Force-directed scheduling* [4]. Schopnosti vývojáře bývají obvykle omezené na nalezení suboptimálního řešení plánování jednoduchých algoritmů.

Nástroj pro automatizovanou transformaci z vyššího programovacího jazyka do jazyka pro popis architektury (*High-level synthesis*) by měl vývojáře do určité míry odstínit od optimalizační problematiky a místo toho nabídnout některá řešení.

Automatizovaný proces transformace (zachycený na obrázku 4) zahrnuje kroky:

1. Překlad modelu procesoru v jazyce ISAC do vnitřní reprezentace. Vnitřní reprezentace zachycuje strukturu procesoru, propojení komponent a abstraktní syntaktické stromy výpočtů behaviorálních sekcí. Další kroky procesu transformace se opakují pro každou operaci jazyka ISAC až po krok Konstrukce objektové reprezentace.
2. Analýza abstraktního syntaktického stromu operace, rozdělení vykonání výpočtů do taktů. Jednotlivé výpočty mohou obecně trvat více taktů. Výstupem této fáze procesu je graf řízení výpočtu. Graf řízení výpočtu po skončení této fáze procesu popisuje sekvenční vykonání výpočtu tak, jak bylo popsáno v jazyku C.
3. Odhalení datových závislostí. Datová závislost vzniká mezi zdrojovými prvky (např. registry) a operacemi (např. sečtení hodnot registrů). Zdrojové prvky a operace představují uzly, datové závislosti tvoří hrany. Výstupem této fáze procesu je graf datových závislostí.
4. Optimalizace řízení výpočtu a přidělení prostředků. V grafu datových závislostí jsou identifikovány nezávislé komponenty. Operace v rámci nezávislých komponent lze provádět paralelně. Kromě prostředků definovaných uživatelem v **`RESOURCE`** sekci existují další implicitně vytvořené prostředky. Pokud se například v jazyku C vyskytne sčítání, v hardwaru bude vytvořena sčítačka (nebo ALU). Je užitečné, aby bylo využití těchto prostředků rovnoměrně rozložené mezi takty výpočtu. Syntetizátor obvodu pak může tyto prostředky použít opětovně v různých taktech, místo aby vytvářel jejich nové instance pro každý takt zvlášť. Výstupem této fáze je optimalizovaný graf řízení výpočtu.
5. Konstrukce objektové reprezentace. Objektová reprezentace popisuje obvod v objektech jazyka C++. Objekty odpovídají jak hardwarovým prostředkům popisujícím architekturu (např. signál, registr, multiplexor, paměť RAM) tak logickým prvkům na vyšší úrovni popisu (např. řídící jednotka). Objektová reprezentace se svým významem podobá vnitřní reprezentaci kódu v překladačích.
6. Výstup do VHDL. Tato fáze procesu představuje tzv. zadní část transformace. Z objektových reprezentací operací jsou vytvořeny entity jazyka VHDL. Nad těmito entitami je *top-level* entita, která je řídí. Hardwarové prostředky popisující architekturu jsou do VHDL převedeny přímo. Logické prvky na vyšší úrovni popisu jsou nejprve převedeny na hardwarové prostředky a následně do VHDL. Mezi další prvky procesoru dodané zvenčí patří například dekodér instrukcí, který se generuje odděleně.
7. Konstrukce simulátoru. K ověření funkcionality obvodu je vhodné provézt tzv. *cycle accurate* simulaci. Simulaci by sice šlo provézt z vygenerovaného kódu v jazyce pro popis architektury (VHDL, viz předchozí krok), časová náročnost takové simulace by ale byla pro složitější obvody neúnosná. Je vhodné, aby se paralelně s tvorbou kódu v jazyce pro popis architektury generoval simulátor ve vyšším programovacím jazyce.



Obrázek 4: Proces transformace

4 Abstraktní syntaktický strom

Po překladu modelu procesoru v jazyku ISAC je pro každou behaviorální sekci vytvořen abstraktní syntaktický strom, který zachycuje strukturu výpočtu. Abstraktní syntaktický strom je kořenový, uzlově ohodnocený, orientovaný, acyklický. Existuje jediný uzel, tzv. kořen stromu, do něhož nevstupuje žádná hrana. Do všech ostatních uzlů grafu vstupuje právě jedna hrana [6]. Z každého uzlu vystupují nejvýše dvě hrany označené jako „left“ a „right“.

Mezi významné uzly patří

- **`SYNTAX_C_AST`** – kořenový uzel. Hrana z tohoto uzlu může vézt do uzlu ***`EXPR`*** (výraz), **`DECLARATIONBLOCK`**, **`STATEMENTLIST`**.
- **`DECLARATIONBLOCK`** – blok deklarací lokálních proměnných. Hrana z **`DECLARATIONBLOCK`** vede do uzlu **`DECLARATIONLIST`**.
- **`DECLARATIONLIST`** – seznam deklarací lokálních proměnných. Hrany z **`DECLARATIONLIST`** vedou do uzlů **`DECLARATION`** nebo **`DECLARATIONLIST`**.

- **DECLARATION** – deklarace proměnné. V levém podřízeném uzlu **TYPE** je typ proměnné, v pravém uzlu **ID** je název proměnné. Jazyk C povoluje deklaraci proměnných se stejným názvem v různých blocích. Při generování abstraktního syntaktického stromu jsou názvy proměnných přejmenovány tak, aby byly jedinečné. Následující kroky transformace nemusejí analyzovat kolize názvů proměnných.
- **STATEMENTLIST** – seznam sekvenčních příkazů. Hrana z tohoto uzlu může vézt do uzlu ***EXPR***, **SELECTIONSTATEMENT** (podmínka), **ITERATIONSTATEMENT** (cyklus), **STATEMENTLIST**.
- ***EXPR*** – výraz. Existuje několik druhů výrazů. Podle druhu výrazu se liší předpony v názvu, např. **ASSIGNMENTEXPR** pro přiřazení výsledku do paměťového prvku. Hrana z uzlu ***EXPR*** může vézt do uzlu **ID**, **CONST**, **DATASTRUCTACCESS**, **FUNCTIONCALL**, ***EXPR***.
- **ID** – identifikátor proměnné nebo prostředku definovaného uživatelem
- **CONST** – konstanta
- **DATASTRUCTACCESS** – přístup ke struktuře pomocí tečkové notace jazyka C, např. „*myRAM.read*“. Hrany z uzlu **DATASTRUCTACCESS** vedou k uzlům ID nebo **DATASTRUCTACCESS**.
- **FUNCTIONCALL** – volání funkce nebo aktivace operace. Hrana z **FUNCTIONCALL** směřuje k argumentům funkce, což jsou uzly ***EXPR*** nebo **ARGUMENTEXPRESSION_LIST**.
- **ARGUMENTEXPRESSION_LIST** – argumenty funkce. Hrana z tohoto uzlu vede k ***EXPR*** nebo **ARGUMENTEXPRESSION_LIST**.
- **SELECTIONSTATEMENT** – podmínka. Podřízené uzly specifikují
  - typ podmínky („*if*“ nebo „*switch*“) v uzlu **COMMAND**,
  - výraz podmiňující vykonání kladného bloku v uzlu **SELECTIONSTATEMENT_CONDITION**,
  - v uzlu **IFTHEN** – kořen kladného bloku, který se vykoná při splnění podmínky,
  - v uzlu **IFELSE** – kořen bloku, který se vykoná při nesplnění podmínky.
- **ITERATIONSTATEMENT** – cyklus. Podřízené uzly specifikují
  - typ cyklu („*for*“ nebo „*while*“) v uzlu **COMMAND**,
  - pro typ cyklu „*for*“ inicializaci proměnné cyklu v uzlu **ITERATIONSTATEMENT_INITIVARIANT**,
  - podmínku vykonání těla cyklu v uzlu **ITERATIONSTATEMENT_CONDITION**,
  - pro typ cyklu „*for*“ je v seznamu příkazů (v těle cyklu) na posledním místě příkaz pro změnu proměnné cyklu v uzlu **ITERATIONSTATEMENT_CHANGEIVARIANT**.

5 Konstrukce grafu řízení výpočtu

Graf řízení výpočtu je orientovaný graf $G=(U,H,p)$. U je konečná množina uzlů, H je konečná množina orientovaných hran. $H=\{(u,v)|u,v \in U\}$. Graf může obsahovat cykly. Uzly reprezentují operace nad lokálními proměnnými nebo zdroji definovanými uživatelem. Hrany reprezentují přechod od jedné operace ke druhé, tedy řízení běhu výpočtu. Jediný uzel p, do kterého nevstupuje žádná hrana, je považován za počáteční.

Z uzlu může vycházen nula až neomezený počet hran:

- Žádná hrana z uzlu nevychází, pokud po operaci v uzlu nenásleduje žádná jiná operace – výpočet končí.
- Jediná hrana z uzlu vychází tehdy, když řízení běhu výpočtu není podmíněno žádnou podmínkou.
- Dvě hrany z uzlu vycházejí tehdy, když je v uzlu vyhodnocována podmínka a na základě výsledku vyhodnocení (pravda, nepravda) se má pokračovat jednou z větví. Hrany z uzlu nesou výsledek vyhodnocení jako podmínku přechodu.
- Více než dvě hrany mohou z uzlu vycházet tehdy, když uzel reprezentuje podmínku typu "*switch*". Hrany z uzlu nesou hodnotu porovnání jako podmínku přechodu. Jedna z hran vždy nese speciální hodnotu "*default*", která zajišťuje, že výpočet bude pokračovat za tělem "*switch*", pokud běh výpočtu nepůjde po žádné podmíněné hraně.

Graf řízení výpočtu má právě jeden uzel, z něhož nevychází žádná hrana. Tento uzel se považuje za koncový. Smysl jediného koncového uzlu bude vysvětlen dále v textu.

Algoritmus konstrukce grafu řízení výpočtu prochází abstraktní syntaktický strom od kořenového uzlu metodou "do hloubky".

Jestliže algoritmus narazí na uzel abstraktního syntaktického stromu, který je typu ***EXPR*** (výraz), pak tento uzel převede na nový uzel grafu řízení výpočtu. Nový uzel si vnitřně udržuje odkaz na uzel abstraktního syntaktického stromu, ze kterého byl vytvořen. Hrana do nového uzlu vede od předchozího nového uzlu. Pokud program v jazyku C neobsahuje cykly a větvení, pak graf řizení výpočtu má tvar řetězu.

Uzel **SELECTIONSTATEMENT** abstraktního syntaktického stromu zastupuje podmíněné vykonání ("*if*" nebo "*switch*"). Tento uzel je převeden na uzly grafu řízení výpočtu:

- uzel, který testuje splnění podmínky a směruje běh výpočtu do některé z větví,
- podgrafy jednotlivých větví, které byly získány rekurzivní analýzou podřízených uzlů rodičovského uzlu **SELECTIONSTATEMENT**,
- uzel, který spojuje řízení výpočtu ze všech větví.

Orientované hrany vedou z uzlu testujícího splnění podmínky do kořenů podgrafů jednotlivých větví. Z koncových uzlů podgrafů větví vedou hrany do uzlu spojujícího řízení výpočtu. U podmíněného vykonání typu „switch" vede jedna přímá nepodmíněná hrana do uzlu spojujícího řízení výpočtu.

Uzel **ITERATIONSTATEMENT** představuje cyklus „*for*" nebo „*while*". Typ cyklu je rozlišen v bezprostředně podřízeném uzlu **COMMAND**. Pokud se jedná o typ cyklu „*while*", pak uzel **ITERATIONSTATEMENT** je převeden na uzly grafu řízení výpočtu:

- uzel, který testuje splnění podmínky cyklu a směruje běh výpočtu do větve zastupující tělo cyklu nebo do uzlu následujícího za cyklem "*while*",
- podgraf větve zastupující tělo cyklu.

Orientovaná hrana vede z uzlu testujícího splnění podmínky do kořene podgrafu těla cyklu. Z koncového uzlu podgrafu těla cyklu vede hrana zpět do uzlu testujícího splnění podmínky.

Cyklus typu „*for*" má oproti cyklu „*while*" navíc

- uzel iniciátoru,

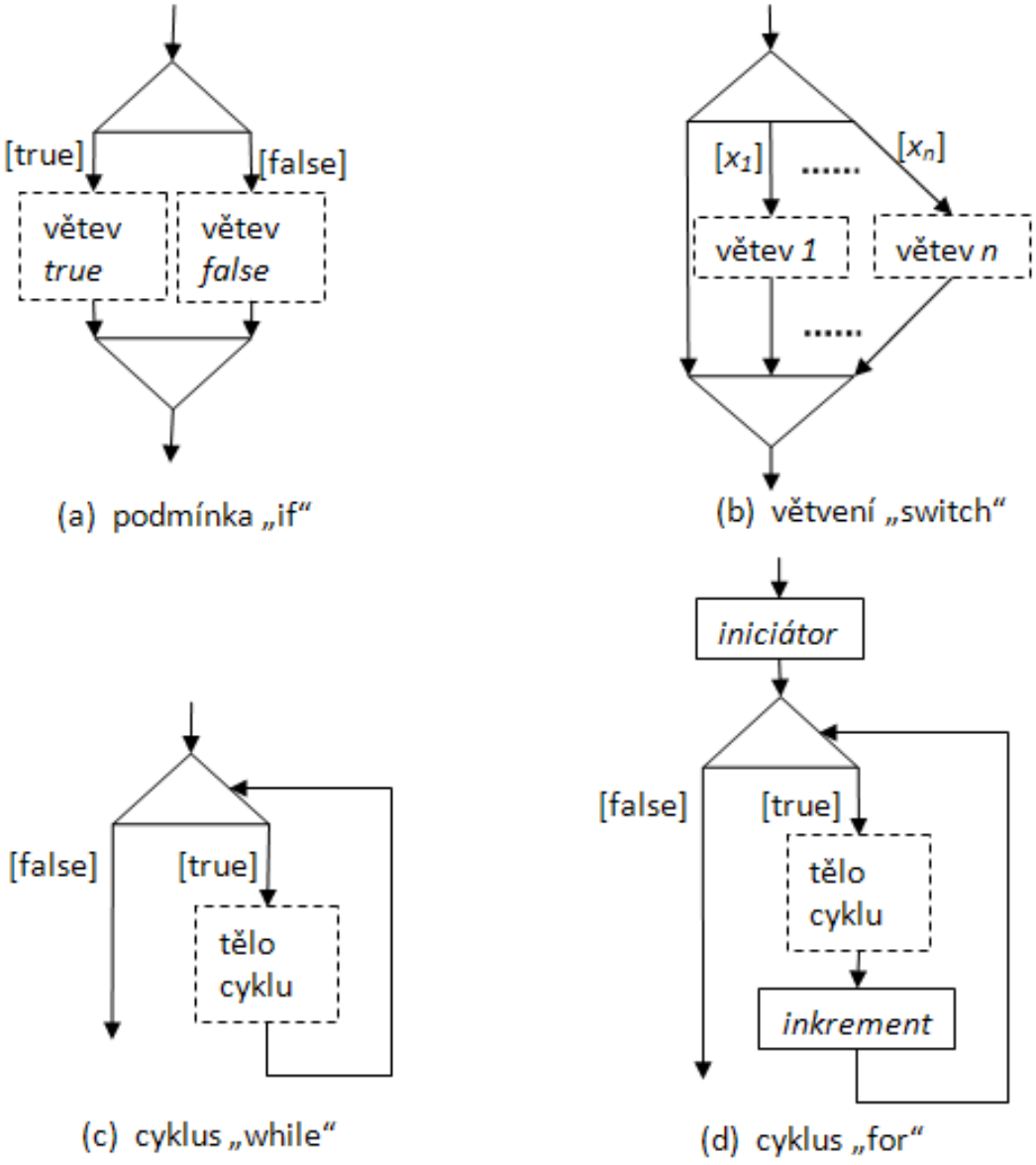
- uzel inkrementu.

Uzel iniciátoru předchází uzlu testování podmínky cyklu – z iniciátoru vede orientovaná hrana do uzlu testování podmínky cyklu. Z uzlu podgrafu těla cyklu vede hrana do uzlu inkementu (místo do uzlu testování podmínky cyklu). Z uzlu inkrementu vede hrana do uzlu testování podmínky cyklu.

Obrázek 5 znázorňuje grafy řízení výpočtu, které byly vytvořeny pro podmínky a cykly.

Uzly **`DECLARATIONBLOCK`** resp. **`DECLARATION`** abstraktního syntaktického stromu zavádějí lokální proměnné. Lokální proměnné jsou následně uloženy do pomocné datové struktury. Na konstrukci grafu řízení výpočtu nemají vliv.

Uzel **`FUNCTIONCALL`** abstraktního syntaktického stromu je převeden na jeden uzel grafu řízení výpočtu (podobně jako by šlo o uzel ***`EXPR`***). Pokud má uzel **`FUNCTIONCALL`** podřízené uzly ***`EXPR`*** – výpočty hodnot argumentů, pak podgrafy odpovídající uzlům ***`EXPR`*** jsou zařazeny před uzlem odpovídajícímu uzlu **`FUNCTIONCALL`**.



Obrázek 5: Příkazy, které způsobí větvení výpočtu

Pro následující kroky transformace je důležité, aby uzly grafu řízení výpočtu měly poznamenánu informaci o číslu základního bloku *(basic block)*, do kterého patří. Velké množství optimalizací kódu je prováděno nad základními bloky [7].

Uzly grafu řízení výpočtu obsahují tyto významné informace:

- název operace,
- číslo základního bloku,
- odkaz na uzel abstraktního syntaktického stromu, na základě něhož byl vytvořen.

6 Konstrukce grafu datových závislostí

Graf datových závislostí je orientovaný graf $G=(U,H)$. U je konečná množina uzlů, H je konečná množina orientovaných hran. $H=\{(u,v)|u,v \in U\}$. Graf datových závislostí nemá počáteční ani koncový uzel a neobsahuje cykly.

Uzly reprezentují

- výpočetní operace (ve smyslu sčítání apod.),
- příkazy ke čtení nebo zápisu z/do lokálních proměnných nebo zdrojů definovaných uživatelem.

Lokální proměnné jsou v hardwaru reprezentovány pomocí registrů nebo pomocí signálů (po optimalizaci). Lokální proměnná je také považována za zdrojový prvek.

Hrany reprezentují datové závislosti mezi uzly operací a zdrojů. Mezi uzly čtení a zápisů z/do zdrojů nevedou přímé hrany.

Cílem konstrukce grafu datových závislostí je nalézt komponenty grafu, které nejsou navzájem spojeny žádnou hranou. Tyto nezávislé komponenty představují části kódu jazyka C, které lze vykonávat paralelně, protože se navzájem datově neovlivňují. Pro paralelně vykonatelné části kódu platí omezení, že musejí patřit do stejného základního bloku.

Algoritmus konstrukce grafu datových závislostí prochází základní bloky grafu řízení výpočtu. Pro každý blok zkonstruuje graf datových závislostí. Algoritmus zapsaný v pseudokódu:

```
for each základní blok b in graf řízení výpočtu G
    nechť Gb je nový graf datových závislostí pro základní blok b
    Gb=(Ub, Hb), Ub jsou uzly, Hb jsou orientované hrany
    Ub = Ø, Hb = Ø
    rcounter = 0, wcounter = 0

    for each uzel u in základní blok b
       /* u je uzel reprezentující operaci uop */
       nechť r1,…,rk jsou datové vstupy (názvy zdrojových prvků)
       operace uop
       nechť w1,…,w1 jsou datové výstupy (názvy zdrojových prvků)
       operace uop
       Ub = Ub U {u}
```

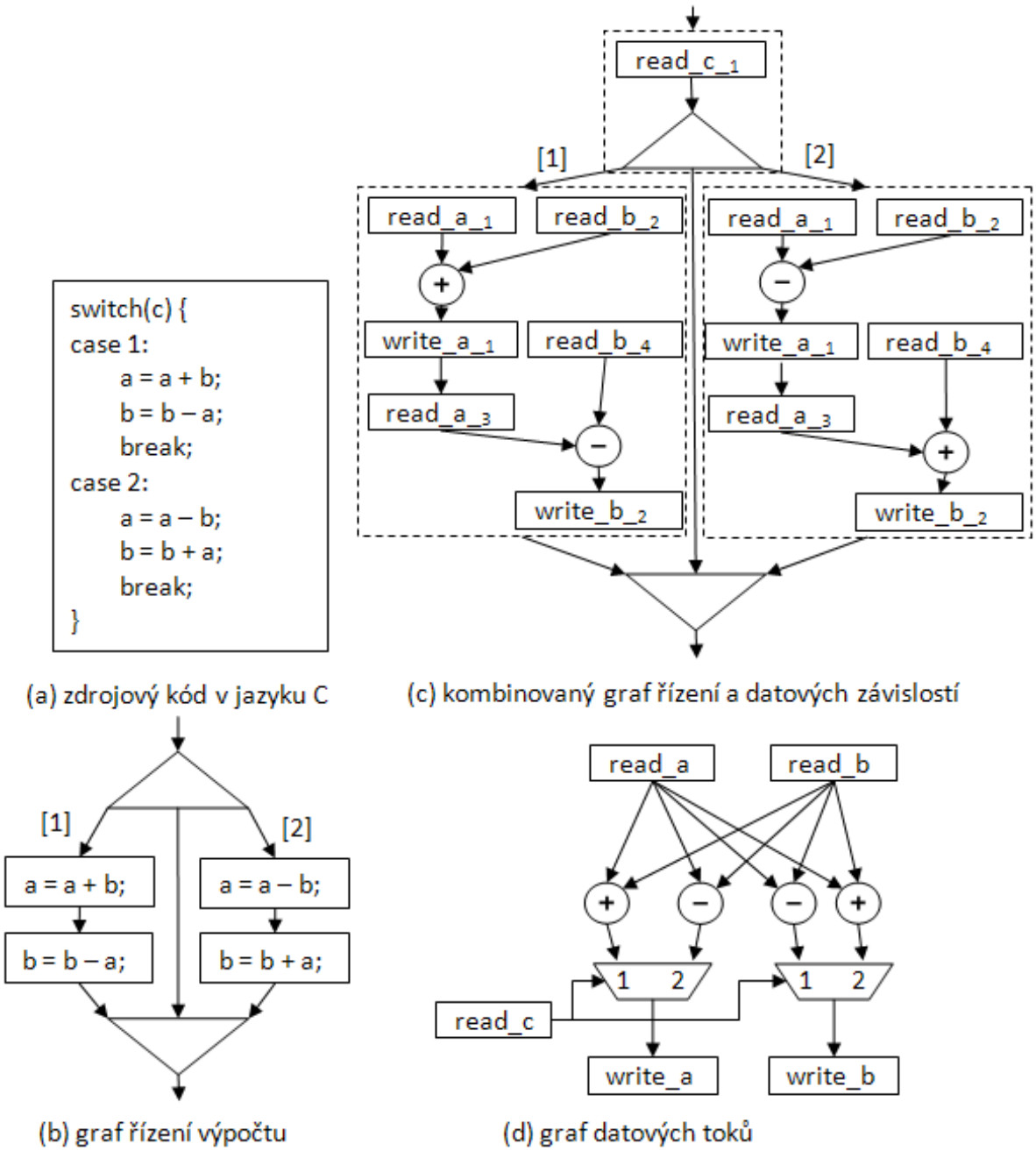
```
        for each datový vstup r_i in {r_1,…,r_k}
            rcounter++
            U_b = U_b U {read_r_i_rcounter}
            h = (read_r_i, u)
            H_b = H_b U {h}
            for each write_r_i_x in U_b
                /* vytvoření závislosti mezi zápisem a čtením */
                H_b = H_b U {(write_r_i_x, read_r_i_rcounter)}
            end for
        end for

        for each datový výstup w_i in {w_1,…,w_l}
            wcounter++
            U_b = U_b U {write_w_i_wcounter}
            h = (u, write_w_i)
            H_b = H_b U {h}
        end for
    end for
    ulož graf G_b k základnímu bloku b
end for
```

Proměnné `rcounter` a `wcounter` slouží ke vzájemnému odlišení čtení a zápisů z/do stejné proměnné.

Po skončení algoritmu má každý základní blok přiřazen graf datových závislostí. Graf řízení výpočtu obohacený o grafy datových závislostí se nazývá kombinovaný graf řízení a datových závislostí (*control/data flow graph* [4]). Obrázek 6 znázorňuje příklad transformace kódu v jazyku C do grafu řízení výpočtu (b), kombinovaného grafu řízení a datových závislostí (c). Graf datových toků (d) představuje popis kódu ve stylu blízkém jazykům pro popis hardwaru.

7 Redukce výšky stromu výrazů

Výpočet matematických výrazů může být značně neefektivní, pokud je prováděn sekvenčně. Redukce výšky stromu výrazů provádí vhodné uzávorkování nad výrazy tak, aby jejich podvýrazy mohly být vyhodnoceny paralelně. Například pravé straně příkazu `x=a+b+c+d*e*f*g+h` by odpovídal efektivní paralelní výpočet při uzávorkování `x=((a+b)+(c+h))+((d*e)*(f*g))`. Přední části překladačů obvykle vracejí neefektivní (sekvenční) variantu. Algoritmy redukce výšky stromu výrazů využívají asociativity, komutativity a distributivity operátorů k přeskládání a uzávorkování výrazů. V následující kapitole je uveden algoritmus redukce výšky stromu výrazů, který je postavený na modifikaci Huffmanova kódování.



Obrázek 6: Příklad transformace zdrojového kódu. Převzato z [4] a upraveno. Obdélník s přerušovanou čárou v grafu (c) ohraničuje základní bloky.

7.1 Huffmanův algoritmus

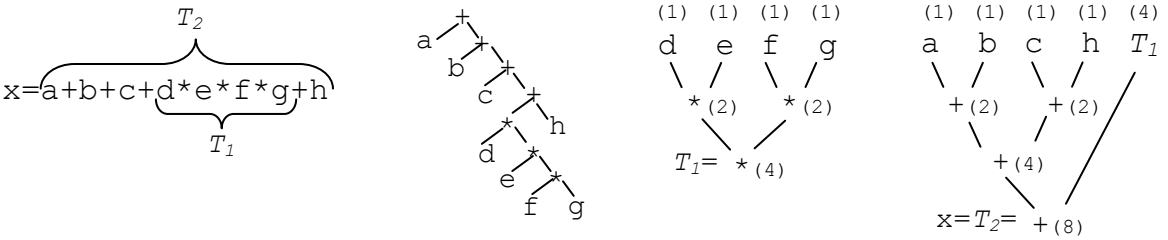
Optimální algoritmus redukce výšky stromu je postaven na modifikaci Huffmanova kódovacího stromu[8]. Optimalitou se rozumí nejmenší možná průměrná hloubka stromu vzhledem k váhám operandů.

Základní varianta Huffmanova algoritmu pracuje takto:

1. přiřadí všem operandům váhu 1,
2. nalezne podvýrazy tvořené stejnými operátory,

3. vyváží podstromy pro komutativní a asociativní operátory tak, že slučuje operandy s nejnižší váhou a váhy sčítá.

Na obrázku 7 je uvedeno srovnání efektivního a neefektivního vyhodnocení výrazu.



(a) příkaz s vyznačenými podvýrazy

(b) neoptimální vyhodnocení

(c) optimální vyhodnocení výrazů

Obrázek 7: Redukce výšky stromu

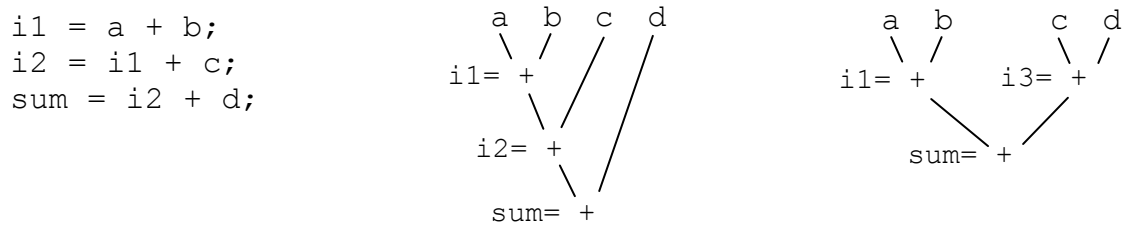
Algoritmus lze snadno modifikovat, aby prioritně prováděl slučování konstant *(constant folding)*. Stačí konstantám nastavit váhu 0 a zbytek algoritmu se postará o to, aby došlo k jejich přednostnímu vyhodnocení. Výraz složený z konstant je vyhodnocen v době kompilace. Na obrázku 8 je uveden příklad slučování konstant. Výraz v přerušovaném rámečku je vyhodnocen prioritně v době kompilace a dále se pracuje pouze s výslednou konstantou (10).

(a) příkaz obsahující konstanty

(b) redukce výšky stromu výrazů společně se slučováním konstant

Obrázek 8: Redukce výšky stromu výrazů společně se slučováním konstant

V předchozích příkladech na obrázcích 7 a 8 byly znázorněny redukce výšky stromů výrazů pro samostatné příkazy. Huffmanův algoritmus dokáže redukovat výšku stromu pro více (vnořených) příkazů najednou. Je potřeba upravit základní variantu algoritmu. Algoritmus tentokrát bude rozlišovat, zda proměnná, do které se zapisuje výsledek, je definovaná uživatelem nebo je dočasná. Uživatelem definovaná proměnná smí nabývat pouze takových hodnot, které odpovídají vyhodnocení výrazů v původním zdrojovém kódu. Dočasné proměnné naproti tomu mohou nabývat libovolných hodnot, případně lze dočasné proměnné rušit či přidávat. Uživatelské proměnné a podvýrazy tvořené stejnými operátory budou dále v textu označovány jako kořeny *(roots)*. Huffmanův algoritmus provede redukci výšky stromů zvlášť pro kořeny a až potom pro výrazy, ve kterých jsou kořeny použity. Volba uživatelských proměnných má vliv na výšku stromu, jak dokládá obrázek 9. Proměnná *i2* na obrázku je v jednom případě (b) uživatelská a ve druhém případě (c) dočasná. V případu (b) musí být *i2* vyhodnocena tak, jak bylo předepsáno ve zdrojovém kódu (a). V případě (b) došlo k nahrazení *i2* za novou dočasnou proměnnou *i3*.



(a) výpočet složený z několika příkazů

(b) proměnné *sum* a *i2* jsou uživatelské

(c) pouze proměnná *sum* je uživatelská

Obrázek 9: Redukce výšky stromu výrazu pro více příkazů najednou

Redukce výšky stromu pracuje nad instrukcemi v základním bloku. Pro výpočet Huffmanova algoritmu je vhodné, aby byly instrukce uloženy ve vnitřní reprezentaci SSA *(Static Single Assignment)*[9]. Každá instrukce v SSA obsahuje cíl uložení výsledku (*R*), operátor (*op*) a dva vstupní operandy (*Ra*, *Rb*). Do každé proměnné (*R*) je výsledek instrukce uložen nejvýše jedenkrát. Pokud je ve zdrojovém kódu ukládána hodnota do proměnné (*R*) několikanásobně, pak kompilátor vytvoří novou verzi proměnné (R_1) a nahradí všechna následující použití proměnné (*R*) za novou verzi (R_1). Někdy nelze v době kompilace určit, která verze proměnné bude použita. V takovém případě vloží kompilátor na místo použití funkci $\varphi(R_1,..,R_n)$, která značí, že správná verze proměnné závisí na běhu programu. Pro každou verzi proměnné se uchovává informace o tom, kde byla definována (tj. instrukce zápisu do proměnné) a seznam použití proměnné (tj. instrukce čtení z proměnné).

Dále v textu budou pro jednoduchost považovány všechny verze proměnných, ze kterých se čte pouze jednou, za dočasné. Toto omezení lze snadno rozšířit.

7.1.1 Huffmanův algoritmus zapsaný v pseudokódu

Algoritmus byl převzat z [8] a upraven.

```
Nalezení a vyvážení kořenů
FindRoots()
{
      Přeskoč neasociativní nebo nekomutativní operátory.
      for each instruction I = 'R <- op, Ra, Rb'
            if op(I) is not associative or not commutative
                  continue;

      Instrukce je kořen:
         A. pokud se z proměnné R čte více než jednou
         B. nebo se z R nečte v žádné instrukci
         C. nebo se z R čte v instrukci s jiným operátorem.

      if (NumUses(R) > 1) or (Use(R)=Ø) or (op(Use(R)) != op(I))
            roots = roots U {I}
            processed(R) = false

      Setřídění seznamu kořenů podle precedence operátorů
      v instrukci od nejvyššího po nejnižší (např. * > +).
```

```
      Sort_op(roots)

      Pro každý kořen se vyvolá funkce BalanceTree pro vyvážení
      podstromu.
      while roots not empty
            I = 'R <- op, Ra, Rb' = Def(Pop(roots))
            if not( I ∈ visited )
                  BalanceTree(I)
}
```

Vyvážení podstromu instrukcí tvořených stejným asociativním a komutativním operátorem. I je instrukce (R_I<-op,Ra,Rb) v kořeni tohoto podstromu.

```
BalanceTree(root I)
{
      Pomocný zásobník uzlů a listů podstromu k procházení.
      worklist: stack
      Fronta listů podstromu setříděných podle váhy
      leaves: priority_queue
      Označení kořene I za navštíveného
      visited = visited U {I}
      Vložení argumentů instrukce I na zásobník
      Push(worklist, Ra)
      Push(worklist, Rb)
      V cyklu se prochází množina uzlů a listů. Listy se přesunou do
      fronty listů (s váhou 1). Uzly jsou buď kořeny a budou
      vyváženy nebo uzly se stejným operátorem jako kořen I a budou
      vloženy na zásobník worklist k procházení.
      while not( worklist is empty )
      {
            T = Pop(worklist)
            T je list, pokud je T proměnná, do které nebyla zapsána
            hodnota (Def(T) vrací instrukci zápisu do prom. T)
            if Def(T) == Ø
                  Enqueue(leaves, T, 1)
            else if T ∈ roots
                  Vyvážení podstromu reprezentovaného kořenem T
                  if not (T ∈ visited)
                        BalanceTree(T)
                  Uložení vyváženého podstromu mezi listy
                  Enqueue(leaves, T, Weight(T))
            else if op(T) == op(I)
                  T je instrukce, jejíž operátor je shodný s kořenem I
                  'R1 <- op1, Ra1, Rb1' = Def(T)
                  Push(worklist, Ra1)
                  Push(worklist, Rb1)
      }
```

```
        Konstrukce vyváženého stromu z listů (podle principu
        Huffmanova kódovacího algoritmu)
        while size(leaves) > 1
        {
              Vyjmutí dvou listů s nejnižší váhou z fronty.
              Ra1 = Dequeue_min(leaves)
              Rb1 = Dequeue_min(leaves)
              Vytvoření nové instrukce T. R1 je nová pomocná proměnná.
              T = 'R1 <- op, Ra1, Rb1'
              Vyhodnocení instrukce T při běhu programu předchází
              vyhodnocení kořenové instrukce I.
              insert T before I
              Váha výsledku = součet vah argumentů
              Weight(R1) = Weight(Ra1) + Weight(Rb1)
              Výsledek je uložen zpátky mezi listy
              Enqueue(leaves, R1, Weight(R1))
        }
        Ve frontě listů zbyl jediný list – nový kořen T
        T = 'R1 <- op, Ra1, Rb1' = Dequeue_min(leaves)
        Původní kořenová instrukce I je vyjmuta ze souboru instrukcí
        remove I = 'R_I <- op, Ra, Rb' from instructions
        Nový kořen převezme název cílové proměnné z původní
        kořenové instrukce I.
        R1 = R_I
}
```

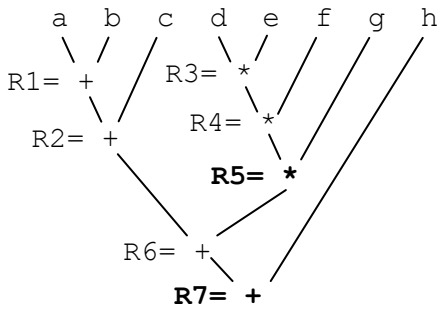
7.1.2 Příklad

Na příkazu `x=a+b+c+d*e*f*g+h` lze ukázat výpočet Huffmanova algoritmu. Na obrázku 10 je znázorněn tento příkaz ve vnitřní reprezentaci SSA a ve formě grafu výpočtu. Tučným písmem jsou zvýrazněny kořenové instrukce. Instrukce `R5 <- *,R4,g` je kořenová, protože cíl přiřazení `R5` je použit v instrukci `R6 <- +,R2,R5` s operátorem `+`, a tedy splňuje podmínku C uvedenou v pseudokódu. Instrukce `R7 <- +,R6,h` je kořenová, protože z `R7` není čteno v žádné instrukci. Tomu odpovídá podmínka B v pseudokódu.

x=a+b+c+d*e*f*g+h

```
R1 <- +,a,b
R2 <- +,R1,c
R3 <- *,d,e
R4 <- *,R3,f
R5 <- *,R4,g
R6 <- +,R2,R5
R7 <- +,R6,h
```

(a) výpočet v reprezentaci SSA



(b) graf výpočtu

Obrázek 10: příklad výrazu ve vnitřní reprezentaci SSA a ve formě grafu.

Seznam kořenů obsahuje dvě instrukce.

```
roots = { 'R5 <- *,R4,g', 'R7 <- +,R6,h' }
```

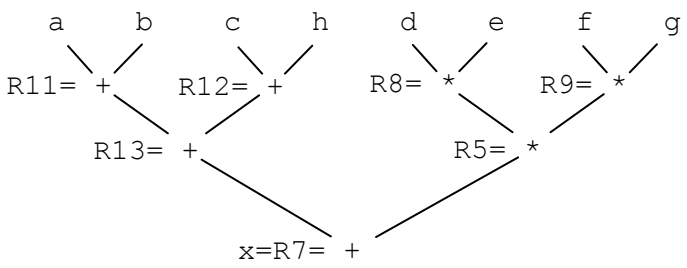
Pro každý kořen je zavolána funkce *BalanceTree*. *BalanceTree* sestaví prioritní frontu listů – dvojic (název proměnné, váha). Algoritmus slučuje proměnné s nejmenší váhou do nových instrukcí, které pak mají váhu rovnu součtu předchozích vah. Slučované proměnné jsou vyznačeny podtržením. Na pravé straně jsou uvedeny nové instrukce.

```
BalanceTree('R5 <- *,R4,g')
      leaves(0) = {(d,1),(e,1),(f,1),(g,1)}        R8 <- *,d,e
      leaves(1) = {(f,1),(g,1),(R8,2)}             R9 <- *,f,g
      leaves(2) = {(R8,2),(R9,2)}                  R10 <- *,R8,R9
      leaves(3) = {(R10,4)}                        R5 <- *,R8,R9

BalanceTree('R7 <- +,R6,h')
      leaves(0) = {(a,1),(b,1),(c,1),(h,1),(R5,4)} R11 <- +,a,b
      leaves(1) = {(c,1),(h,1),(R11,2),(R5,4)}     R12 <- +,c,h
      leaves(2) = {(R12,2),(R11,2),(R5,4)}         R13 <- +,R12,R11
      leaves(1) = {(R13,4),(R5,4)}                 R14 <- +,R13,R5
      leaves(1) = {(R14,8)}                        R7 <- +,R13,R5
```

Obrázek 11 zobrazuje optimalizovaný výpočet po skončení běhu Huffmanova algoritmu. Graf výpočtu je optimálně vyvážený.

```
R8  <- *,d,e
R9  <- *,f,g
R5  <- *,R8,R9
R11 <- +,a,b
R12 <- +,c,h
R13 <- +,R12,R11
R7  <- +,R13,R5
```

(a) výpočet v reprezentaci SSA



(b) graf výpočtu

Obrázek 11: optimalizovaný výpočet v reprezentaci SSA a v odpovídajícím grafu výpočtu

8 Redukce doby výpočtu pro asociativní operátory

V kapitole o redukci výšky stromu výpočtu bylo ukázáno, jak optimálně paralelizovat vyhodnocení výrazu, jehož operátory jsou zároveň asociativní a komutativní. Díky komutativitě šlo přeskládat operandy. Uvažujme operátory, které jsou asociativní a nejsou komutativní. Patří mezi ně například násobení matic. Problém u násobení matic je ten, že nevhodným uzávorkováním operací (tj. nevhodnou volbou pořadí provádění násobení) dojde k vysokému počtu skalárních součinů a součtů. Úlohou optimalizujícího překladače je nalézt takové uzávorkování, aby počet primitivních operací byl

nejmenší. Jednou z možných tříd algoritmů pro nalezení optimálního uzávorkování jsou algoritmy dynamického programování. Na násobení matic bude ukázán jejich princip.

8.1 Násobení matic

Uvažujme násobení dvou matic $C_{(i,k)} = A_{(i,j)} * B_{(j,k)}$. K výpočtu jedné buňky výsledné matice *C* je potřeba vynásobení vektoru *(1,j)* matice *A* a vektoru *(j,1)* matice *B*. Provede se tedy *j* násobení a *j-1* sčítání skalárů. Výsledná matice *C* má *i*k* buněk, proto počet násobení skalárů je *i*j*k*. Zavedeme funkci

$$\text{cenaNásobeníMatic}(A_{(i,j)}, B_{(j,k)}) = i*j*k,$$

která udává cenu, kterou je potřeba „zaplatit“ za vynásobení matic *A* a *B*. Cena koresponduje s náročností provedení operace. Cílem dynamického programování je nalézt takové pořadí násobení matic, že celková cena bude minimální.

Na příkladu násobení matic $Y_{(5,2)}=A_{(5,3)}*B_{(3,4)}*C_{(4,10)}*D_{(10,2)}$ lze ukázat, že volba pořadí (uzávorkování) násobení ovlivňuje celkový počet násobení skalárů, tedy celkovou cenu. Analyzujme cenu násobení matic při dvou možných uzávorkováních:

$$
\begin{aligned}
Y_{(5,2)} &= ((A_{(5,3)}*B_{(3,4)})*C_{(4,10)})*D_{(10,2)} \\
&= (AB_{(5,4)}*C_{(4,10)})*D_{(10,2)} \\
&= ABC_{(5,10)}*D_{(10,2)} \\
\text{cena} &= \text{cenaNásobeníMatic}(A_{(5,3)}, B_{(3,4)}) + \\
&\quad \text{cenaNásobeníMatic}(AB_{(5,4)}, C_{(4,10)}) + \\
&\quad \text{cenaNásobeníMatic}(ABC_{(5,10)}, D_{(10,2)}) \\
&= 5*3*4 + 5*4*10 + 5*10*2 \\
&= \underline{\underline{360}}
\end{aligned}
$$

$$
\begin{aligned}
Y_{(5,2)} &= (A_{(5,3)}*B_{(3,4)})*(C_{(4,10)}*D_{(10,2)}) \\
&= AB_{(5,4)}*CD_{(4,2)} \\
\text{cena} &= \text{cenaNásobeníMatic}(A_{(5,3)}, B_{(3,4)}) + \\
&\quad \text{cenaNásobeníMatic}(C_{(4,10)}, D_{(10,2)}) + \\
&\quad \text{cenaNásobeníMatic}(AB_{(5,4)}, CD_{(4,2)}) \\
&= 5*3*4 + 4*10*2 + 5*4*2 \\
&= \underline{\underline{180}}
\end{aligned}
$$

V druhém případě jsme obdrželi řešení s poloviční výpočetní náročností.

Algoritmus dynamického programování postupuje takto:

1. Postupně závorkuje všechny sousední matice do skupin po dvou. Pro každou dvoučlennou skupinu (závorku) vypočítá cenu násobení. Například pro *Y=A*B*C*D* vypočítá cenu násobení pro výrazy *(A*B), (B*C), (C*D)*.
2. Postupně závorkuje všechny sousední matice do skupin po třech. Pro *Y=A*B*C*D* máme dvě tříčlenné skupiny: *(A*B*C)* a *(B*C*D)*. Násobení tříčlenné skupiny lze rozdělit na násobení dvoučlenné skupiny a jedné matice. Například pro skupinu *(A*B*C)* lze násobit *((A*B)*C)* nebo *(A*(B*C))*. Ceny násobení dvoučlenných skupin jsou předpočítány z předchozího

kroku. Algoritmus zvolí takové uzávorkování, které vyžaduje nejnižší cenu násobení celé tříčlenné skupiny.

3. Algoritmus závorkuje vždy o 1 delší skupiny matic než v předchozím kroku. Skončí, až když velikost skupiny je rovna počtu matic. Ceny násobení menších skupin jsou vždy předpočítány z předchozích kroků.

8.1.1 Algoritmus dynamického programování

Algoritmus v pseudokódu byl převzat z [10].

```
Prvek pole p[i] udává počet řádků i. matice. Počet sloupců i.
matice=počet řádků řádků (i+1). matice.
Matrix-Chain(array p[1..n]) {
    Prvek s[i,j]=k říká, že součin matic Mi*…*Mj je uzávorkován
    takto: (Mi*…*Mk)*(Mk+1*…*Mj)
    array s[1..n-1,2..n];
    Matice m uchovává minimální ceny násobení matic Mi*…*Mj = m[i,j]
    array m[1..n,1..n];
    Inicializace m: cena m[i,i]=0
    for i = 1 to n do m[i,i] = 0;
    Iterace přes všechny možné delky uzávorkování L ∈ <2, n>
    for L = 2 to n do {
        Iterace přes všechny skupiny délky L. Proměnná i udává
        začátek skupiny. Proměnná j udává konec skupiny.
        for i = 1 to n-L+1 do {
            j = i + L - 1;
            Minimální cena násobení matic Mi*…*Mj je neznámá
            m[i,j] = INFINITY;
            Iterace přes všechna rozdělení skupiny na dvě části.
            Proměnná k udává index rozdělení.
            for k = i to j-1 do {
                Cena násobení (Mi*…*Mk)*(Mk+1*…*Mj)
                q = m[i, k] + m[k+1, j] + p[i-1]*p[k]*p[j]
                Pokud bylo nalezeno lepší rozdělení než předchozí, pak
                se uloží cena a index nového rozdělení
                if (q < m[i, j]) {
                    m[i,j] = q;
                    s[i,j] = k;
                }
            }
        }
    }
    V m[1,n] je cena nejlepšího uzávorkování a v s[1,n] je index
    rozdělení součinu Mi*…*Mj. Další rozdělení je v s[1, s[1,n]]
    a v s[s[1,n]+1, n] atd. rekurzivně.
    return (m,s);
}
```

Složitost algoritmu je $O(n^3)$, kde n je počet matic. Existuje i efektivnější algoritmus běžící v čase $O(n*log(n))$[11].

8.1.2 Příklad

Mějme opět násobení matic $A_{(5,3)}*B_{(3,4)}*C_{(4,10)}*D_{(10,2)}$. Ukážeme postup výpočtu algoritmu dynamického programování.

Algoritmus vypočítá cenu násobení všech dvojic:

Dvojice:	$AB_{(5,4)}$	$BC_{(3,10)}$	$CD_{(10,2)}$
Cena:	m[1,2] = 60	m[2,3] = 120	m[3,4] = 80

Algoritmus vypočítá cenu násobení všech trojic:

Trojice:	$ABC_{(5,10)}$		$BCD_{(3,2)}$	
Možná rozdělení:	A(BC)	(AB)C	(BC)D	B(CD)
Cena rozdělení:	60+200 = 260	120+150= 270	120+60 = 180	80+24 = 104
Nejlepší rozdělení:	A(BC)		B(CD)	
Cena:	m[1,3] = 260		m[2,4] = 104	

Algoritmus vypočítá cenu násobení všech čtveřic:

Čtveřice:	$ABCD_{(5,2)}$	
Možná rozdělení:	(ABC)D	A(BCD)
Cena rozdělení:	260 + 100 = 360	104 + 30 = 134
Nejlepší rozdělení:	A(BCD)	
Cena:	m[1,4] = 134	

Algoritmus vypočítal optimální posloupnost násobení matic při uzávorkování A*(B*(C*D)). Cena tohoto maticového násobení je 134 skalárních násobení.

8.2 Závěr

Ukázali jsme, že lze optimalizovat vyhodnocení výrazů složených z asociativních operátorů, pokud existuje funkce ceny vyhodnocení podvýrazu. Pokud je v době kompilace známý rozměr matic, pak lze optimální vyhodnocení násobení vypočítat předem (staticky). V opačném případě je potřeba do programu doplnit kód pro optimalizaci, která proběhne za běhu (dynamicky). Pod pojmem program si lze představit i obvodovou reprezentaci algoritmu v hardwaru.

9 Optimalizace distributivních operátorů

Výrazy složené z distributivních operátorů lze matematickými úpravami převézt na výrazy s nižším počtem operací (např. násobení). Mají-li dva výrazy společné podvýrazy, pak se využije distributivity

k vytknutí těchto podvýrazů před závorku. Při realizaci obvodu pro výpočet výrazu je důležité, aby byl minimalizován počet drahých funkčních jednotek – v tomto případě násobiček.

Příklad výrazu: $12x^4y^3 + 10xy^2 + 6x^2y^2 = 2xy^2(6x^3y + 5 + 3x)$.
Vytknutím došlo ke snížení počtu násobení z původních 14 na 9.

Cílem je nalezení největšího společného dělitele mezi sčítanými výrazy. Největší společný dělitel dvou a více výrazů je součin všech společných prvočinitelů jednotlivých výrazů[12]. Proměnné x,y v příkladu jsou literály a budou považovány za prvočinitele.

Rozklad příkladu na prvočinitele s vyznačením společných prvočinitelů je:
$\underline{2}*2*3*\underline{x}*x*x*x*\underline{y}*\underline{y}*y + \underline{2}*5*\underline{x}*\underline{y}*\underline{y} + \underline{2}*3*\underline{x}*x*\underline{y}*\underline{y} = 2xy^2(6x^3y + 5 + 3x)$.

Rozkladu přirozených čísel na součin prvočísel se lze vyhnout. Největší společný dělitel dvou čísel lze efektivně získat např. Euklidovým algoritmem.

Aby byl počet operací (násobení) snížen na minimum, nemusí být vždy výhodné počítat největší společný dělitel pro všechny výrazy. Například ve výrazu $y + 5x^2y + 10x^2$ je největším společným dělitem všech podvýrazů číslo 1 a při vyhodnocování se provede 5 operací násobení. Pokud ale vytkneme z posledních dvou podvýrazů ($5x^2y$ a $10x^2$) největší společný dělitel ($5x^2$), tak získáme výraz $y + 5x^2(y+2)$, při jehož vyhodnocení se provedou pouze 3 operace násobení. Původní výraz lze upravit i do tvaru $y(1 + 5x^2) + 10x^2$. Tento poslední výraz vyžaduje 5 operací násobení, a tedy nepřináší žádné vylepšení. Je potřeba navrhnout algoritmus transformace původního výrazu na nový, který bude obsahovat minimální počet operací násobení.

9.1 Algoritmus transformace

Nejprve je ukázáno, že algoritmus transformace původního výrazu na nový, který bude obsahovat minimální počet operací násobení, bude nutně mít vysokou časovou složitost. Následně bude navržen jeden takový algoritmus a analyzována jeho časová složitost.

Definice: Relace podobnosti, oblast [14]
Nechť A je množina. Binární relace $\rho \subseteq A \times A$ se nazývá relací podobnosti, jestliže

1. $\forall a \in A : (a,a) \in \rho$, tj. ρ je reflexivní
2. $\forall a,b \in A : (a,b) \in \rho \Rightarrow (b,a) \in \rho$, tj. ρ je symetrická

Podmnožina $B \subseteq A$ se nazývá oblast relace podobnosti, jestliže

1. $\forall a,b \in B : (a,b) \in \rho$, tj. ρ je úplnou relací na B
2. $\forall a \in A : a \notin B \Rightarrow \exists b \in B : (a,b) \notin \rho$, tj. B je maximální podmnožina, na které je ρ úplná

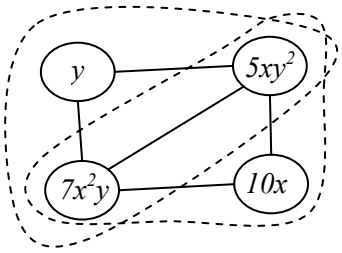
Nechť A je množina výrazů. Nechť φ je relace nad množinou A taková, že

1. $\forall a \in A : (a,a) \in \varphi$, tj. φ je reflexivní
2. $\forall a,b \in A :$ největší společný dělitel $a, b \neq 1$, pak $(a,b) \in \varphi$.

Věta: Relace φ nad množinou výrazů A je relací podobnosti.

Důkaz: Reflexivita plyne z definice. Symetrie platí, neboť $\forall a,b \in A : (a,b) \in \varphi \Leftrightarrow$ největší společný dělitel $a, b \neq 1 \Leftrightarrow$ největší společný dělitel $b, a \neq 1 \Leftrightarrow (b,a) \in \varphi$.

Nad množinou výrazů lze zkonstruovat graf relace podobnosti. Uzly představují jednotlivé výrazy. Hrany spojují uzly, které mají největší společný dělitel různý od 1. V grafu je dále možné vyznačit oblasti relace podobnosti. Oblasti jsou úplné podgrafy grafu, neboť oblast je maximální podmnožina, na které je relace podobnosti úplná. Obrázek 12 znázorňuje graf relace podobnosti pro výrazy $y+5xy^2+7x^2y+10x$. Přerušované čáry ohraničují oblasti relace podobnosti: {y, $5xy^2$, $7x^2y$}, {$5xy^2$, $7x^2y$, $10x$}.



Obrázek 12: Graf relace podobnosti s vyznačenými oblastmi

Má-li nějaký algoritmus nalézt pro daný výraz ekvivalentní výraz s minimálním počtem násobení (metodou vytýkaní společného dělitele), pak nutně musí analyzovat všechny oblasti relace podobnosti, protože každá oblast představuje potenciální možnost pro snížení počtu násobení – z výrazů v oblasti lze vytknout společného dělitele. Identifikovat všechny oblasti je ovšem NP-úplný problém. Stačí si uvědomit, že problém nalezení maximálního úplného podgrafu (tedy kliky) je NP-plný[15].

9.1.1 Návrh algoritmu

Algoritmus vychází z některých obecných znalostí o největším společném děliteli (*gcd*)[13]:

1. komutativita: *gcd(a,b) = gcd(b,a)*,
2. asociativita: *gcd(a, gcd(b,c)) = gcd(gcd(a,b), c)*,
3. gcd pro více výrazů: *gcd(a,b,c) = gcd(a, gcd(b,c))*

Algoritmus projde všechny možné kombinace vytýkání největších společných dělitelů nad všemi výrazy. Vytýkání společného dělitele pro více výrazů je rozděleno na sérii vytýkání společných dělitelů dvou výrazů. Na pořadí výrazů nezáleží díky komutativitě největšího společného dělitele.

```
Algoritmus si pamatuje řešení s nejmenším počtem násobení.
int nejmenší_počet_násobení = INFINITY;
array nejlepší_řešení;

Rekurzivní funkce hledá řešení s nejmenším počtem násobení.
transformace(array výrazy[1..n])
{
   Zjistí počet násobení, pokud se neprovede žádné vytýkání.
   int pn = počet_násobení(výrazy);
   if(pn < nejmenší_počet_násobení)
   {
```

```
        nejmenší_počet_násobení = pn;
        nejlepší_řešení = výrazy;
    }
    Projde všechny dvoučlenné kombinace výrazů. Z dvojice výrazů
    (i,j) vytkne jejich největší společný dělitel (gcd_ij). Pokud je
    gcd_ij různý od 1, pak se sestaví nové pole výrazů, pro které
    se rekurzivně vypočítá nové řešení transformace.
    for(i=1; i<n; i++)
    {
        for(j=i+1; j ≤ n; j++)
        {
            Funkce gcd vypočítá největší společný dělitel
            gcd_ij = gcd(výrazy[i], výrazy[j]);
            if(gcd_ij != 1)
            {
                Nový výraz ze dvou výrazů a společného dělitele. Např.
                pro 5x^2 a 10x je nový_člen = 5x(x+2).
                nový_výraz = gcd_ij * vytkni(gcd_ij, výrazy[i],
                    výrazy[j]);
                Pole nových výrazů obsahuje původní výrazy mimo
                ty na idexech i, j. Pole navíc obsahuje nový_výraz.
                nové_výrazy = array[sčítanci[1..(i-1)],
                                    sčítanci[(i+1)..(j-1)],
                                    sčítanci[(j+1)..n],
                                    nový_výraz];
                Volání transformace pro menší pole výrazů.
                transformace(nové_výrazy);
            }
        }
    }
}
```

Algoritmus využívá pomocné funkce, které lze snadno zkonstrukovat:

- `(int) počet_násobení(array výrazy[1..n]);` vrátí počet operací násobení ve výrazech. Například pro pole výrazů [y, $5x^2y$, $10x^2$], jimž odpovídá výraz $y + 5x^2y + 10x^2$, vrátí číslo 5. Složitost funkce je $O(n)$, protože musí projít všechny prvky pole.
- `(výraz) gcd(výraz_1, výraz_2);` vrátí největší společný dělitel dvou výrazů. Funkce rozloží výrazy na prvočinitele a vrátí společný podvýraz. Výraz v závorce považuje za prvočinitel. Například výrazy $10xy^2(z^2+1)$, $6x^2y^2(z^2+1)(p+q)$ rozloží na $2*5*x*y*y*(z^2+1)$, $2*3*x*x*y*y*(z^2+1)*(p+q)$ a vrátí $2xy^2(z^2+1)$. Složitost funkce je $O(1)$ vzhledem k počtu výrazů.
- `(výraz) vytkni(gcd_12, výraz_1, výraz_2);` vrátí výraz v závorce odpovídající vytknutí společného dělitele `gcd_12` z výrazů `výraz_1`, `výraz_2`. Například pro `gcd_12`$=2xy^2(z^2+1)$, `výraz_1`$=10xy^2(z^2+1)$, `výraz_2`$=6x^2y^2(z^2+1)(p+q)$ vrátí výraz $(5+3x(p+q))$. Složitost funkce je $O(1)$ vzhledem k počtu výrazů.

9.1.2 Příklad

Běh algoritmu je ilustrován na následujícím příkladu. Najděme ekvivalentní výraz k výrazu $y + 5xy^2 + 7x^2y + 10x$, aby počet násobení byl minimální. Nejlepší řešení bude uloženo v poli `nejlepší_řešení` po dokončení běhu funkce `transformace([y, 5xy^2, 7x^2y, 10x])`.

Řádek odpovídá volání funkce `transformace`. Tučným písmem jsou vyznačeny nové výrazy. Odsazení znázorňuje hierarchii rekurzivního zanoření. Napravo je vyhodnocení počtu operací násobení v daném výrazu na řádku. Minimální počet násobení je orámován.

[y, 5xy^2, 7x^{2}y, 10x]	pn=7
[**y(1+5xy)**, 7x^{2}y, 10x]	pn=7
[**y((1+5xy)+7x^2)**, 10x]	pn=6
[y(1+5xy), **x(7xy+10)**]	pn=7
[**y(1+7x^2)**, 5xy^2, 10x]	pn=7
[**y((1+7x^2)+5xy)**, 10x]	pn=6
[y(1+7x^2), **5x(y^2+2)**]	pn=6
[y, **xy(5y+7x)**, 10x]	pn=5
[**y(1+x(5y+7x))**, 10x]	pn=5
[y, **x(y(5y+7x)+10)**]	pn=4
[y, **5x(y^2+2)**, 7x^{2}y]	pn=6
[**y(1+7x^2)**, 5x(y^2+2)]	pn=6
[y, **x(5(y^2+2)+7xy)**]	pn=5
[y, 5xy^2, **x(7xy+10)**]	pn=6
[**y(1+5xy)**, x(7xy+10)]	pn=6
[y, **x(5y^2+(7xy+10))**]	pn=5

Algoritmus nalezl ekvivalentní výraz $y + x(y(5y+7x)+10)$, pro jehož vyhodnocení je zapotřebí pouze čtyř násobení.

9.1.3 Analýza složitosti

Analyzujme složitost algoritmu vzhledem k počtu výrazů v poli. Algoritmus při každém volání funkce `transformace` provede zjištění počtu operací násobení se složitostí $O(n)$, $\binom{n}{2}$ volání funkce `gcd` a maximálně $\binom{n}{2}$ volání funkce vytkni, sestavování nového pole výrazů a rekurzivního volání funkce `transformace`. Sestavení nového pole má složitost n. Každé volání funkce `transformace` nad polem n výrazů může způsobit až $\binom{n}{2}$ rekurzivních volání funkce `transformace` nad polem $(n-1)$ výrazů. Maximální počet volání funkce `transformace` je $O\binom{n}{2} * O\binom{n-1}{2} * \ldots * O(1) = O(n^2) * O((n-1)^2) * \ldots * 1$. Složitost výpočtu algoritmu je velká, a proto jej lze aplikovat pouze na omezené pole výrazů.

9.2 Závěr

Byl navržen algoritmus pro transformaci výrazů na výrazy s minimálním počet násobení. Algoritmus lze upravit, aby minimalizoval nebo maximalizoval i jinou funkci nad výrazy, nejen počet operací násobení. Algoritmus má nepříznivou časovou složitost. Není obtížné navrhnout jiný algoritmus s heuristikou a lepší časovou složitostí, avšak s vyšším počtem operací násobení.

Pro některé výrazy existuje rozklad na součiny (faktory), kde celkový počet násobení je menší než řešení nalezené výše navrženým algoritmem. Jedná se například o výraz $x^2 + 2xy + y^2$, pro který je známé řešení faktorizace $(x+y)^2$, zatímco navrhovaný algoritmus nalezne pouze $x(x+2y)+y^2$ nebo $x^2 + y(2x+y)$.

10 Plánování

Algoritmus transformace má po skončení kroku Konstrukce grafu datových závislostí k dispozici kombinovaný graf řízení výpočtu a datových závislostí (*control/data flow graph, CDFG* [4]). Algoritmus plánování musí na základě tohoto grafu naplánovat zahájení a ukončení operací, čtení/zápisů z/do zdrojových prvků.

Vstupem algoritmu plánování je kombinovaný graf řízení výpočtu a datových závislostí. Před spuštěním algoritmu plánování se všem uzlům grafu doplní hodnota latence. Latence udává počet taktů potřebných k výpočtu operace nebo načtení/zapsání dat z/do zdrojových prvků. Hodnoty latence jsou získány

- od uživatele – uživatel specifikuje latence čtení a zápisu ve zdrojovém popisu procesoru,
- ze specifikace prvků cílové architektury,
- z obecně známých standardů – například latence čtení z registru je vždy jeden takt.

Problém u specifikace prvků cílové architektury je v tom, že knihovna prvků může obsahovat několik prvků se stejnou funkcionalitou, ale různou latencí a různou prostrovou náročností na čipu. Například sčítačka může být řešena rychlým, ale drahým obvodem „*carry-look-ahead adder*“, nebo pomalejším a levnějším „*carry-ripple adder*“. Plánovací algoritmus by mohl rozhodnout, že na kritické cestě výpočtu použije rychlý prvek, zatímco na nekritické cestě použije levnější prvek[4]. Nabízí se i řešení založené na spolupráci člověka-návrháře, který tento problém rozhodne. Rozhodnutí člověka může například vycházet z analýzy výkonnostního profilu algoritmu případně ze simulace obvodu. Pro jednoduchost budeme uvažovat, že z knihovny prvků budeme vybírat vždy nejrychlejší prvek. Jeho latenci zapíšeme do atributu „latence“ k uzlům grafu, které prvek používají.

Výstupem algoritmu plánování je nový graf řízení výpočtu, který se od předchozí verze grafu řízení výpočtu liší v tom, že:

- obsahuje nové uzly, které zahajují nebo ukončují čtení/zápis z/do zdrojových prvků,
- všechny uzly obsahují nový atribut „číslo taktu“, který udává pořadí vykonání operace.

Uzly, které mají shodnou hodnotu atributu „číslo taktu“, jsou vykonány ve stejném taktu paralelně. Paralelně lze vykonat operace, které nemají datové závislosti.

Prvky architektury mají kromě latencí ještě další charakteristiky, např. velikost na čipu nebo spotřebu energie. Tato práce se zabývá dvěma směry plánování [4]:

- plánování omezené časem *(time-constrained scheduling)* – minimalizace počtu funkčních jednotek při zadaném maximálním počtu taktů,
- plánování omezené prostorem *(resource-constrained scheduling)* – minimalizace počtu taktů při zadaném maximálním počtu funkčních jednotek.

10.1 Návaznost čtení a zápisu na operaci

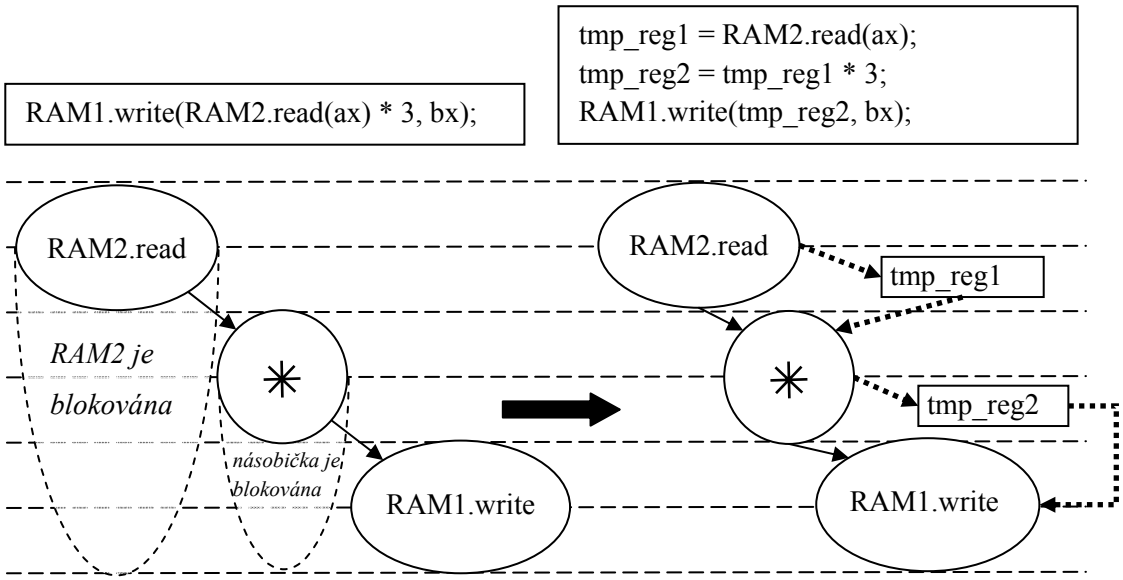
Každá atomická výpočetní operace očekává, že při spuštění obdrží vstupní data, která se po dobu výpočtu nebudou měnit. Podobně se očekává, že v průběhu zápisu do zdrojového prvku jsou zapisovaná data konstantní. Mezi operací čtení nebo zápisu a výpočetní operací vzniká těsný vztah: výpočetní operace musí navazovat na čtení, zápis musí navazovat na výpočetní operaci. U některých zdrojových prvků je těsný vztah na škodu a může být výhodné se jej zbavit.

Aby bylo možné posoudit, zda je vhodné zbavovat se těsného vztahu, je potřeba analyzovat vstupy a výstupy operací. Datové vstupy a výstupy operace lze rozdělit do dvou skupin podle toho, zda vstup/výstup lze provézt v jednom taktu nebo více taktech. Příkladem zdrojového prvku, k jehož datům lze přistupovat v jednom taktu, je registr. Dále v textu budeme za registr považovat synchronní D-klopný obvod. Do synchronního registru lze navíc ve stejném taktu zapisovat nová data a zároveň z něj číst původní data. U obecného zdrojového prvku je potřeba počítat s horšími vlastnostmi. Příkladem obecného zdrojového prvku je paměť RAM. Uživatel ve zdrojovém souboru jazyku ISAC specifikuje latence čtení a zápisu. U pamětí bývají latence obvykle vícetaktové.

Způsob, jakým se lze zbavit těsného vztahu, je zavedení pomocných registrů mezi čtení a výpočetní operaci, výpočetní operaci a zápis nebo mezi dvě výpočetní operace. Pomocný registr vkládáme tehdy, když nechceme, aby byl některý prostředek blokovaný déle než je nezbytně nutné. Místo blokovaného prostředku bude blokován pomocný registr. Díky pomocným registrům lze k prostředku přistoupit dřív nebo později, podle toho, jak o tom rozhodne algoritmus plánování. U sdílených prostředků (např. pamětí RAM nebo aritmeticko-logických jednotek) vede tento přístup ke zvýšení výkonnosti celého systému.

Na stejném principu je založeno řetězení výpočtu *(pipelining)*. Rozdíl spočívá v tom, že u řetězených výpočtů sestavuje linku řetězení návrhář systému explicitně, zatímco při sestavování obvodu z vyššího programovacího jazyka rozhoduje o mezistupních zpracování plánovač.

Obrázek 13 znázorňuje řešení problému těsné návaznosti vstupně-výstupních operací na výpočetní operaci (násobení). Prvotní návrh (a) způsobí, že paměť RAM2 a násobička budou blokovány, dokud neskončí zápis do paměti RAM1. Návrh (b) zavádí pomocné registry mezi operace.



(a) operace na sebe těsně navazují a prostředky jsou blokovány

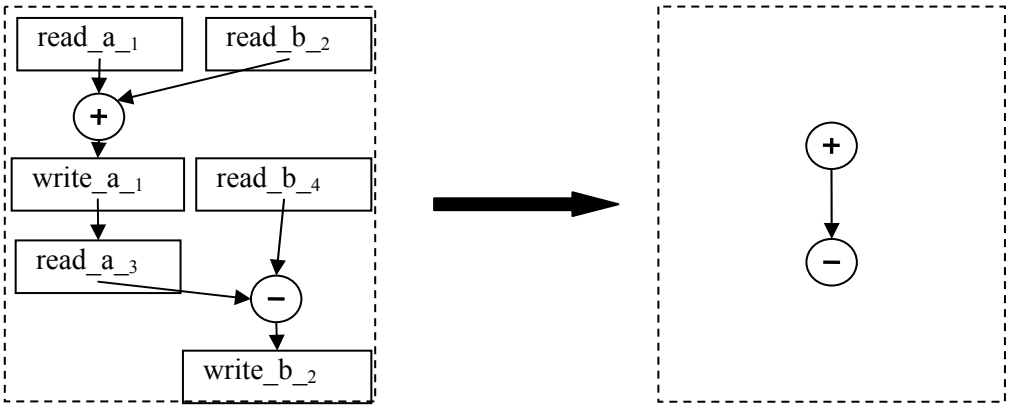
(b) těsné vztahy jsou odstraněny díky pomocným registrům

Obrázek 13: Příklad řešení těsné návaznosti vstupně-výstupních operací na výpočetní operaci (násobení).

10.2 Redukovaný graf datových závislostí

Předpokládejme, že jsme v předchozím kroku odstranili těsné vztahy mezi operacemi. Máme tedy graf, ve kterém se (nepravidelně) střídají výpočetní operace se vstupně/výstupními. Většinu vstupně/výstupních operací tvoří přístupy do registrů. Latence přístupů do registrů je vždy 1. Čtení i zápis do registru je navíc vždy překryt s výpočetní operací. Z tohoto důvodu můžeme množinu uzlů grafu datových závislostí redukovat o uzly čtení a zápisu z/do registrů. Pokud vedla hrana *h* od operace op_i k zápisové operaci do registru reg_j a hrana *g* od operace čtení z registru reg_j do operace op_k, pak bude do množiny hran přidána nová hrana $l=(op_i, op_j)$. Hrany od výpočetních operací do zápisových operací registrů a hrany od čtecích operací registrů do výpočetních operací jsou odstraněny (tj. hrany *h* a *g*).

Obrázek 14 znázorňuje příklad redukce grafu. Předpokládá se, že všechny vstupně výstupní operace jsou mezi registry.



(a) neredukovaný graf (b) redukovaný graf

Obrázek 14: Redukce grafu

10.3 Algoritmus plánování ASAP

Algoritmus ASAP naplánuje vykonání operací co nejdřív je to možné (odtud zkratka ASAP – *as soon as possible*). Vstupem algoritmu je redukovaný graf datových závislostí pro jeden základní blok kódu (v jazyku C). Výstupem je redukovaný graf datových závislostí, kde každý uzel (tj. operace) obsahuje číselný atribut *E* (z angl. *earliest* – nejdřívější). Atribut *E* udává číslo taktu, ve kterém může být operace zahájena nejdříve.

Nevýhodou algoritmu ASAP je to, že předpokládá neomezené množství dostupných výpočetních prostředků (např. sčítaček, komparátorů, ALU apod.). Existují algortimy, které zohledňují počet výpočetních prostředků. Tyto algoritmy využívají výstup algoritmu ASAP resp. hodnotu atributu *E*.

Agoritmus ASAP v pseudokódu (převzato z [4] a upraveno):
Vstup/Výstup: $G = (U, H)$
nechť $U' = U$

```
for each uzel u_i ∈ U' do
      if Pred(u_i) = Ø then
            E_i = 1;
            U' = U' - {u_i};
      else
            E_i = 0;
      end if
end for

while U' ≠ Ø do
      for each uzel u_i ∈ U' do
            if ALL_NODES_SCHEDULED_E(Pred(u_i)) then
                  E_i = max_{p∈Pred(u_i)}(E_p + latence_p);
                  U' = U' - {u_i};
            end if
      end for
```

```
end while
```

Funkce *Pred(u_i)* vrátí množinu uzlů, ze kterých vede přímá orientovaná hrana do uzlu u_i. Funkce *max* vrátí nejbližší takt, ve kterém jsou dokončeny všechny předchozí operace. Funkce *ALL_NODES_SCHEDULED$_E$(Pred(u_i))* vratí true, pokud všechny uzly z množiny *Pred(u_i)* mají nenulovou hodnotu atributu *E*.

V prvním cyklu *for* jsou všem uzlům, do kterých nevede žádná hrana, nastaveny atributy *E* na hodnotu 1 a tyto uzly jsou vyjmuty z dalšího zpracování. Cyklus *while* prochází přes všechny nezpracované uzly. V každé iteraci je naplánován takt všem uzlům, jejichž všichni předchůdci (podle funkce *Pred*) jsou naplánování (mají nenulovou hodnotu atributu *E*). Naplánované uzly jsou vyjmuty z dalšího zpracování. Jelikož je graf datových závislostí acyklický, tak algoritmus ASAP v každé iteraci while cyklu naplánuje alespoň jeden uzel. Množina uzlů je konečná, a proto algoritmus vždy skončí. Každý uzel je naplánován na nejbližší možný takt, protože je mu přiřazen takt bezprostředně následující po skončení předchozí operace, na které datově závisí.

10.3.1 Příklad

Algoritmy plánování budou ilustrovány na příkladu transformace algoritmu HAL[16]:

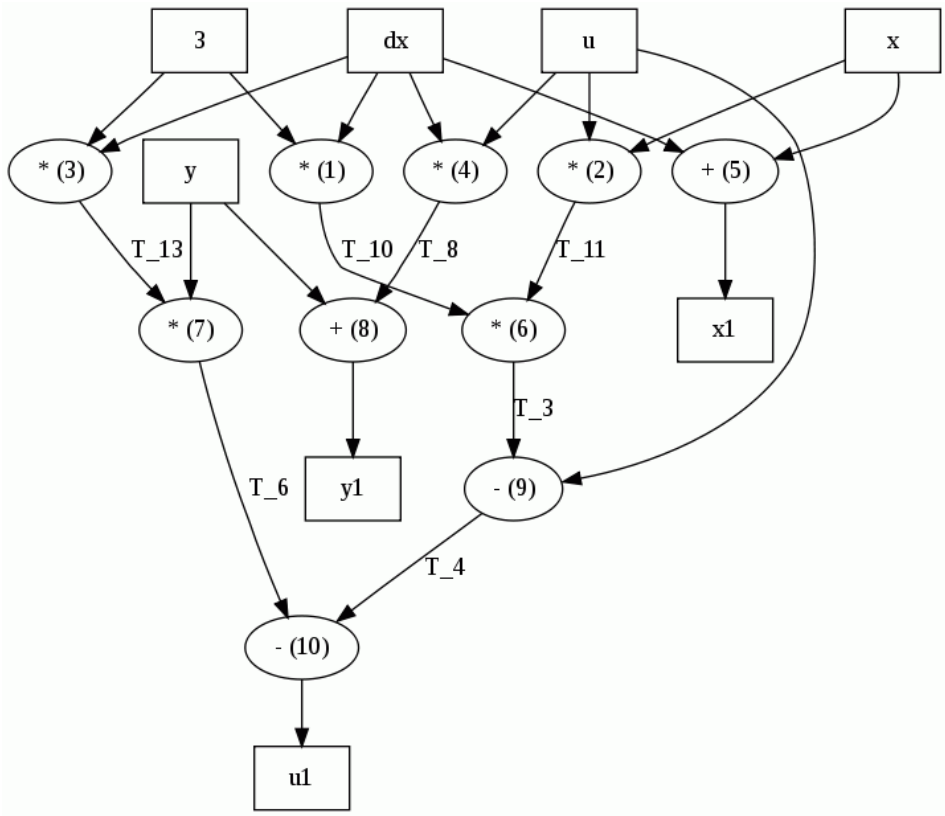
```
x1 := x + dx;
u1 := u - (3*x*u*dx) - (3*y*dx);
y1 := y + (u*dx);
```

Následuje algoritmus HAL zapsaný v interní reprezentaci SSA. Vlevo je tvar SSA, který vrátí překladač (metodou zdola nahoru), v pravo je tvar SSA po optimalizaci Huffmanovým algoritmem. T_*n* označuje pomocné registry. Tučným písmem jsou vyznačeny optimalizované operace. Číslo v závorce udává index operace. (Index operace je použit v obrázcích.)

Neoptimalizovaný zápis	Po optimalizaci Huffmanovým algoritmem	
x1 <- + [x] [dx]	(1)	**T_10 <- * [dx] [3]**
T_1 <- * [3] [x]	(2)	**T_11 <- * [u] [x]**
T_2 <- * [T_1] [u]	(3)	T_13 <- * [dx] [3]
T_3 <- * [T_2] [dx]	(4)	T_8 <- * [u] [dx]
T_4 <- - [u] [T_3]	(5)	x1 <- + [x] [dx]
T_5 <- * [3] [y]	(6)	**T_3 <- * [T_10] [T_11]**
T_6 <- * [T_5] [dx]	(7)	T_6 <- * [y] [T_13]
u1 <- - [T_4] [T_6]	(8)	y1 <- + [y] [T_8]
T_8 <- * [u] [dx]	(9)	T_4 <- - [u] [T_3]
y1 <- + [y] [T_8]	(10)	u1 <- - [T_4] [T_6]

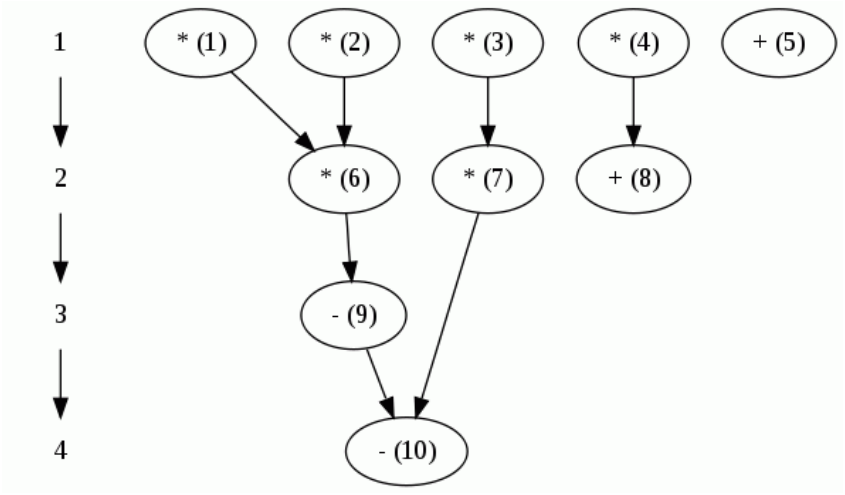
Algoritmy plánování mají na vstupu graf toku dat a naplánují vykonání operací do určitých taktů. Předtím, než jsou plánovací algoritmy spuštěny, lze provézt transformace založené na znalostech vlastností operátorů použitých v operacích. Například násobení `(3*x*u*dx)` v přiřazení `u1 := u - (3*x*u*dx) - (3*y*dx);` je vhodné uzávorkovat pomocí Huffmanova algoritmu do tvaru `((3*x)*(u*dx))`. Obrázek 15 zobrazuje graf toku dat algoritmu HAL po provedení optimalizace

Huffmanovým algoritmem. Obdélníky označují datové registry. Elipsy obsahují operace. Hrany mezi operacemi jsou označeny pomocnými registry, přes které si operace předávají dočasná data.



Obrázek 15: Graf toku dat algoritmu HAL

Obrázek 16 znázorňuje výstup algoritmu plánování ASAP. Na levé straně se nachází čítač taktů. Operace jsou zahájeny v taktu daném atributem *E* (*earliest*, co nejdříve).



Obrázek 16: Graf operací naplánovaných do taktů algoritmem ASAP

10.4 Algoritmus plánování ALAP

Algoritmus plánování ALAP naplánuje vykonání operací co nejpozději je to možné (odtud zkratka ALAP – *as late as possible*). Vstupem algoritmu je redukovaný graf datových závislostí pro jeden základní blok kódu (v jazyku C), stejně jako u algoritmu ASAP. Navíc je potřeba algoritmu dodat časové omezení *T*. Výstupem je redukovaný graf datových závislostí, kde každý uzel (tj. operace) obsahuje číselný atribut *L* (z angl. *latest* – nejpozdější). Atribut *L* udává číslo taktu, ve kterém může být operace zahájena nejpozději, aby byla dokončena do časového limitu *T*.

Algoritmus ALAP je (stejně jako alg. ASAP) necitlivý k počtu dostupných výpočetních prostředků. Výstup algoritmu slouží jako vstup plánovacích algoritmů. Rozdíl hodnot *E* a *L* uzlu grafu udává prostor, ve kterém může být operace naplánována.

Algoritmus ALAP v pseudokódu (převzato z [4] a upraveno):
Vstup/Výstup: G = (U, H)
nechť U' = U

```
for each uzel u_i ∈ U' do
      if Succ(u_i) = Ø then
            L_i = T - latence_i;
            if (L_i < 0 or L_i < E_i) then
                  error(„Zvolena příliš nízká konstanta T.");
            end if
            U' = U' - {u_i};
      else
            L_i = 0;
      end if
end for

while U' ≠ Ø do
      for each uzel u_i ∈ U' do
            if ALL_NODES_SCHEDULED_L(Succ(u_i)) then
                  L_i = min_{p∈Succ(u_i)}(L_p);
                  L_i = L_i - latence_i;
                  if (L_i < 0 or L_i < E_i) then
                        error(„Zvolena příliš nízká konstanta T.");
                  end if
                  U' = U' - {u_i};
            end if
      end for
end while
```

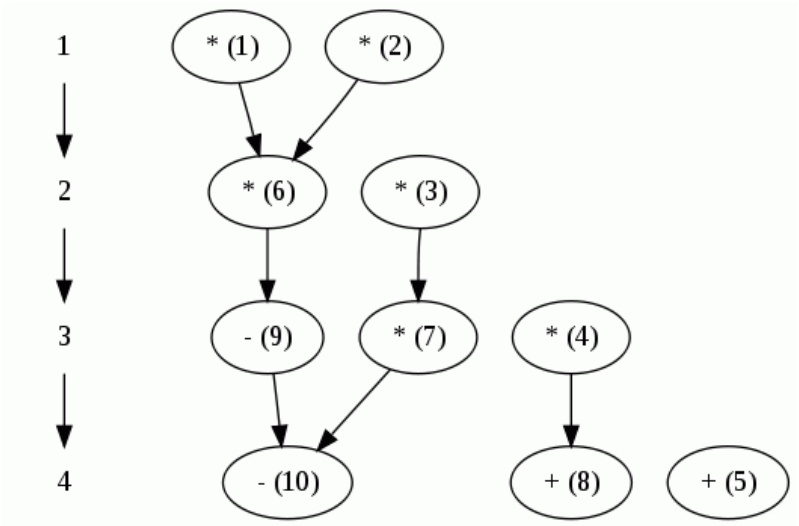
Funkce $Succ(u_i)$ vrátí množinu uzlů, do kterých vede přímá orientovaná hrana z uzlu u_i. Funkce *min* vrátí nejnižší takt, ve kterém jsou naplánovány všechny datově závislé operace z množiny $Succ(u_i)$. Funkce $ALL\_NODES\_SCHEDULED_L(Succ(u_i))$ vrátí true, pokud jsou naplánovány všechny datově závislé operace z množiny $Succ(u_i)$. Algoritmus skončí chybou, pokud by mělo dojít k naplánování operace do záporného čísla taktu. Podobně skončí chybou, pokud by operace měla být naplánována do dřívějšího taktu, než určil algoritmus ASAP. Obě chyby mají původ ve volbě příliš nízké

konstanty *T*. Prakticky nemá smysl volit konstantu *T* nižší než celková doba výpočtu algoritmem ASAP.

V prvním cyklu *for* jsou naplánovány operace, na něž datově nezávisí žádné operace. Ostatní operace jsou naplánovány v cyklu *while*.

10.4.1 Příklad

Algoritmu ALAP je na vstup dán kód algoritmu HAL. Obrázek 17 znázorňuje výstup algoritmu plánování ALAP. Operace jsou zahájeny v taktu daném atributem *L* (*latest*, co nejpozději).



Obrázek 17: Graf operací naplánovaných do taktů algoritmem ALAP

Díky atributům operací *E* a *L* získaných algoritmy ASAP a ALAP je možné vyčíslit rozsah taktů, v nichž může být daná operace zahájena. Tato informace bude využita u pokročilejších algoritmů plánování.

takt	Operace									
	*(1)	*(2)	*(3)	*(4)	+(5)	*(6)	*(7)	+(8)	–(9)	–(10)
1.										
2.										
3.										
4.										

Tabulka 1: Rozsah taktů, ve kterých může být daná operace zahájena

10.5 Celočíselné lineární programování

Lineární programování *(linear programming, LP)* je metoda hledání minima lineární funkce[17]. Problém, v jehož kontextu je minimum hledáno, je popsán soustavou lineárních nerovnic. Koeficienty lineární funce a nerovnic jsou reálná čísla.

Každý lineární program lze zapsat jako:

- Minimalizuj hodnotu funkce $c^T x$
- nad všemi vektory $x \in R^n$ splňujícími $Ax \leq b$

kde $c^T x = c_1 x_1 + \ldots + c_n x_n$ je lineární funkce, A je matice $m \times n$ reálných čísel a $c \in R^n$, $b \in R^m$.

Zvláštní skupinu problémů lineárního programování tvoří úlohy, pro které je požadováno celočíselné řešení $x \in Z^n$. Tato třída problému se nazývá celočíselné lineární programování *(integer linear programming, ILP)*. Pomocí celočíselného lineárního programování lze zformulovat úlohu plánování minimálního nutného počtu funkčních jednotek (např. násobiček nebo ALU) pro zkonstruování obvodu, který provádí výpočet daného algoritmu v minimálním čase[18]. Minimální počet funkčních jednotek implikuje jejich maximální využití. Úlohu lze tedy vnímat jako maximalizaci využití funkčních jednotek. Důležitým aspektem úlohy je také to, že čas (v taktech) pro výpočet daného algoritmu má být minimální. Minimální možný čas na provedení celého výpočtu je dán časem vypočteným algoritmem ASAP – je to atribut *E* nejpozději zahájené operace, formálně $min_T = max\{E_i \mid op_i \text{ je operace}\}$. Minimální čas je známý předem a algoritmus celočíselného lineárního programování je jím omezen.

Třída algoritmů plánování, které jsou předem omezeny časem (počtem taktů), se nazývají časově omezené plánování (*time-constrained scheduling*)[4]. Obvody navržené touto metodou bývají nasazovány do systému běžících v reálném čase. Například u multimediálních obvodů v kamerách je známý čas na zpracování jednoho vzorku obrazu (rámce). Jelikož čas na zpracování jednoho vzorku obrazu je fixní, pak hlavním cílem optimalizace je minimalizování ceny hardware.

10.5.1 Formulace plánování jako úlohy celočíselného lineárního programování

Nechť jsou pro jednoduchost stanoveny dva předpoklady:

1. latence všech operací je jeden takt,
2. datové cesty nejsou řetězené.

Předpokládejme, že je dán graf datových závislostí nad operacemi algoritmu, který má být optimalizován. Graf obsahuje n operací o_i, $1 \leq i \leq n$. Relace precedence mezi operacemi je značena symbolem šipky: $o_i \rightarrow o_j$, pokud operace o_i je přímým předchůdcem operace o_j. Pro každou operaci o_i je známý čas nejdřívějšího možného zahájení E_i (ASAP) a nejpozdějšího možného zahájení L_i (ALAP). Graficky je rozsah taktů, ve kterých může být daná operace zahájena, zobrazen v tabulce 1.

Předpokládejme, že jsou k dispozici různé funkční jednotky. Bývá obvyklé, že jedna funkční jednotka dokáže provádět několik typů operací. Například ve funkční jednotce ALU lze vyhodnocovat sčítání, odčítání nebo porovnávání. Typ funkční jednotky je značen t_i, $1 \leq i \leq m$, kde m je počet typů funkčních jednotek a cena této jednotky (například v jednotkách velikosti čipu) je c_{ti}. Množina FU_{ti} obsahuje všechny operace o_j, které jsou vykonávány funkční jednotkou typu t_i. Formálně $FU_{ti} = \{o_j \mid t_i = typ(o_j),\ 1 \leq j \leq n\}$.

Značení bude demonstrováno na algoritmu HAL (viz kapitola 10.3.1):

```
x1 := x + dx;
```

```
u1 := u - (3*x*u*dx) - (3*y*dx);
y1 := y + (u*dx);
```

Předpokládejme, že jsou k dispozici funkční jednotky: ALU, násobička. Cena ALU=1, násobička=5. Potom

- $m = 2$,
- t_1 … ALU (sčítání, odčítání),
- t_2 … násobička,
- $c_{t1} = 1$,
- $c_{t2} = 5$.

Celkem je 10 operací, 6 x násobení (operace {1,2,3,4,6,7}), 2 x sčítání (operace {5,8}) a 2 x odčítání (operace {9,10}). Potom

- $FU_{t1} = \{o_5, o_8, o_9, o_{10}\}$,
- $FU_{t2} = \{o_1, o_2, o_3, o_4, o_6, o_7\}$.

Zbývá ještě dodefinovat proměnné, jejichž hodnoty jsou řešením úlohy celočíselného programování:

1. M_{ti} – počet funkčních jednotek typu t_i potřebných k provedení výpočtu,
2. $x_{i,j}$ – celočíselné proměnné nabývající hodnot 0 nebo 1 asociované s operací o_i takto:
   - $x_{i,j} = 1$, pokud zahájení operace o_i bude naplánováno do taktu j,
   - $x_{i,j} = 0$ jinak.

Formulace úlohy celočíselného programování [18]:

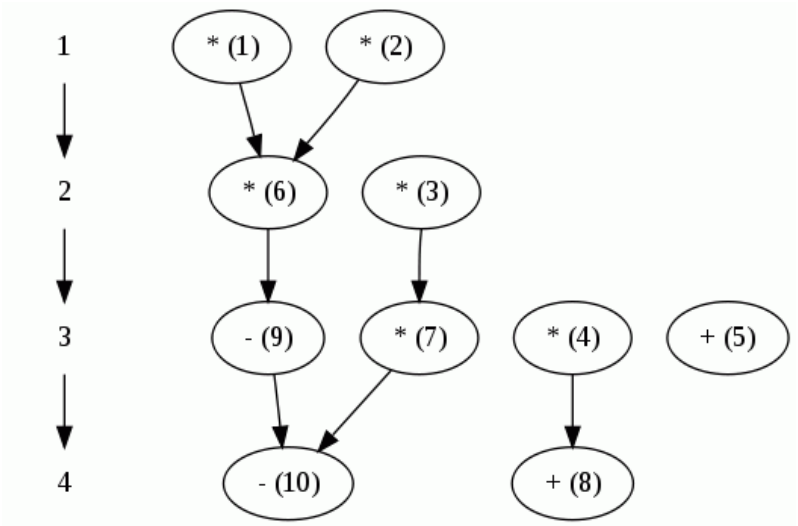
1. Minimalizuj hodnotu funkce $\sum_{i=1}^{m} c_{ti} * M_{ti}$. Funkce vyjadřuje celkovou cenu použitých funkčních jednotek.
2. Omezení $\sum_{\substack{i=1 \\ o_i \in FU_{th}}}^{n} x_{i,j} - M_{tk} \le 0,$ pro $1 \le j \le \min_T, 1 \le k \le m$ říká, že v každém taktu výpočtu (j) má být použito maximálně M_{tk} funkčních jednotek typu t_k. Uvažujme opět příklad algoritmu HAL a rozsah taktů v tabulce 1. V prvním taktu lze provézt pouze jednu operaci sčítaní (5). Proto odpovídající nerovnice omezení pro tento bod je $x_{5,1} - M_{t1} \le 0$, kde t_1 je funkční jednotka ALU. Podobně je sestavena nerovnice pro počet násobiček použitelných v prvním taktu algoritmu: $x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} - M_{t2} \le 0$, neboť pouze operace 1, 2, 3, 4 mohou v prvním taktu využít násobení.
3. Omezení $\sum_{j=E_i}^{L_i} x_{i,j} = 1$ pro $1 \le i \le n$ říká, že každá operace s indexem i smí být zahájena pouze v taktech v rozsahu $j \in < E_i, L_i >$. Například operace o_3 smí být zahájena v taktu 1 nebo 2, a proto odpovídající omezení je $x_{3,1} + x_{3,2} = 1$.
4. Poslední omezení zachycuje vazby mezi operacemi: $\sum_{j=E_i}^{L_i} j * x_{i,j} - \sum_{j=E_k}^{L_k} j * x_{k,j} \le -1$ pro všechny relace precedence mezi operacemi $o_i \rightarrow o_k$. Operace o_i musí proběhnout předtím, než začne operace o_k. Číslo -1 za nerovností říká, že operace o_k smí začít 1 takt po zahájení

operace o_i. Jedná se tedy o latenci operace o_i, kterou jsme pro jednoduchost omezili na 1 takt. Například pro relaci $o_7 \rightarrow o_{10}$ dostaneme omezení $2 * x_{7,2} + 3 * x_{7,3} - 4 * x_{10,4} \leq -1$.

Formulace úlohy celočíselného programování pro příklad HAL:

Funkce/nerovnice	Popis
$5 * M_{MUL} + M_{ALU}$	Minimalizovaná funkce ceny.
$x_{5,1} - M_{ALU} \leq 0$ $x_{5,2} + x_{8,2} - M_{ALU} \leq 0$ $x_{5,3} + x_{8,3} + x_{9,3} - M_{ALU} \leq 0$ $x_{5,4} + x_{8,4} + x_{10,4} - M_{ALU} \leq 0$	Omezení počtu jednotek ALU v jednotlivých taktech (podle bodu 2).
$x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} - M_{MUL} \leq 0$ $x_{3,2} + x_{4,2} + x_{6,2} + x_{7,2} - M_{MUL} \leq 0$ $x_{4,3} + x_{7,3} - M_{MUL} \leq 0$	Omezení počtu násobiček v jednotlivých taktech (podle bodu 2).
$x_{1,1} = 1$ $x_{2,1} = 1$ $x_{3,1} + x_{3,2} = 1$ $x_{4,1} + x_{4,2} + x_{4,3} = 1$ $x_{5,1} + x_{5,2} + x_{5,3} + x_{5,4} = 1$ $x_{6,2} = 1$ $x_{7,2} + x_{7,3} = 1$ $x_{8,2} + x_{8,3} + x_{8,4} = 1$ $x_{9,3} = 1$ $x_{10,4} = 1$	Omezení, ve kterých taktech smí být daná operace zahájena (podle bodu 3).
$1 * x_{3,1} + 2 * x_{3,2} - 2 * x_{7,2} - 3 * x_{7,4} \leq -1$ $1 * x_{4,1} + 2 * x_{4,2} + 3 * x_{4,3} - 2 * x_{8,2} - 3 * x_{8,3} - 4 * x_{8,4} \leq -1$ $2 * x_{7,2} + 3 * x_{7,3} - 4 * x_{10,4} \leq -1$	Omezení vycházející z vazeb mezi operacemi ($o_i \rightarrow o_k$, bod 4). Pokud lze obě operace vykonat pouze v jediném taku (tj. $E_i = L_i \wedge E_k = L_k$), pak není potřeba nerovnici pro operace $o_i \rightarrow o_k$ vytvářet.

Získané optimální řešení je $x_{1,1} = x_{2,1} = x_{3,2} = x_{4,3} = x_{5,3} = x_{6,2} = x_{7,3} = x_{8,4} = x_{9,3} = x_{10,4} = 1$, ostatní $x_{i,j} = 0$. Počet jednotek ALU = 2, násobiček = 2. Obrázek 18 znázorňuje získané řešení ve formě grafu.



Obrázek 18: Graf operací naplánovaných do taktů algoritmem celočíselného lineárního programování

10.5.2 Vícetaktové operace

Zadání úlohy celočíselného lineárního programování lze rozšířit o operace, jejichž latence je větší než jedent takt[18]. Je potřeba upravit omezení č. 2 a 4:

- (omezení 2'): $\sum_{\substack{i=1 \\ o_i \in FU_{tk}}}^{n} \sum_{p=0}^{latence_i - 1} x_{i,j-p} - M_{tk} \leq 0$ pro $1 \leq j \leq \min_T, 1 \leq k \leq m$.

Omezení říká, že v každém taktu výpočtu (j) má být použito maximálně M_{tk} funkčních jednotek typu t_k. Smysl rozšíření je demonstrován na příkladu. Předpokládejme například, že operace o_1 má latenci 2 a $E_1 = L_1 = 1$. Operace o_2 má latenci 2 a $E_1 = L_1 = 2$. Nechť jsou operace o_1 a o_2 vykonávány na stejném typu funkční jednotky (t_1). Tabulka 2 graficky znázorňuje danou situaci.

j	operace o_1 / o_2	Omezení pro daný takt j
1	$x_{1,1}$	$x_{1,1} - M_{t1} \leq 0$
2	$x_{1,1}$ / $x_{2,2}$	$x_{1,1} + x_{2,2} - M_{t1} \leq 0$
3	$x_{2,2}$	$x_{2,2} - M_{t1} \leq 0$

Tabulka 2: Operace o_1 s latencí 2

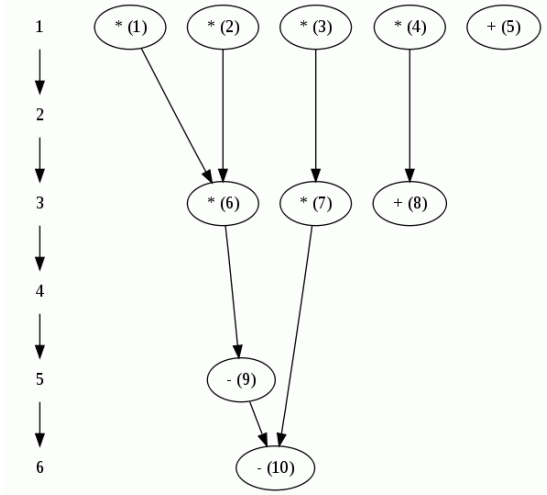
Jestliže má být operace o_1 zahájena v taktu 1, tj. $x_{1,1} = 1$, pak odpovídající funkční jednotka bude blokována i v taktu 2. V nerovnici popisující omezení pro 2. takt proto figuruje i proměnná $x_{1,1}$. Podobně při zahájení operace o_2 v taktu 2, tj. $x_{2,2} = 1$, bude funkční jednotka blokována i v taktu 3. Proměnná $x_{2,2}$ tedy přibyde do nerovnice pro 3. takt. Obě operace se vykonávají na stejném typu funkční jednotky t_1. Protože je známo, že $x_{1,1} = x_{2,2} = 1$, pak z nerovnice pro 2. takt plyne $M_{t1} \geq 2$. To je korektní závěr, protože běhy operací o_1 a o_2 se překrývají, a proto musejí běžet na dvou různých funkčních jednotkách.

- (omezení 4'): $\sum_{j=E_i}^{L_i} j * x_{i,j} - \sum_{j=S_k}^{L_k} j * x_{k,j} \leq -latence_i$ pro všechny relace precedence mezi operacemi $o_i \rightarrow o_k$.

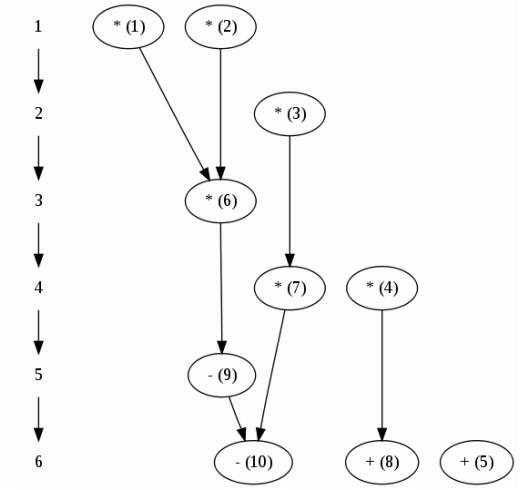
Původní omezení č. 4 počítalo s latencí operace o_i 1 takt. Omezení 4' je jeho zobecněním.

10.5.2.1 Příklad

Mějme algoritmus HAL z kapitoly 10.3.1. Nechť operace násobení potřebuje 2 takty k dokončení výpočtu. V důsledku toho vzniká potřeba přepočítat rozsahy povolených zahájení operací ($E_i - L_i$). Obrázky 19 a 20 zobrazují plánování operací s vícetaktovou latencí algoritmy ASAP a ALAP.



Obrázek 19: Plánování vícetaktových operací algoritmem ASAP



Obrázek 20: Plánování vícetaktových operací algoritmem ALAP

Tabulka 3 znázoňuje rozsahy taktů, ve kterých může být operace zahájena (šedou barvou). Orámování ohraničuje takty, v nichž operace může blokovat odpovídající funkční jednotku.

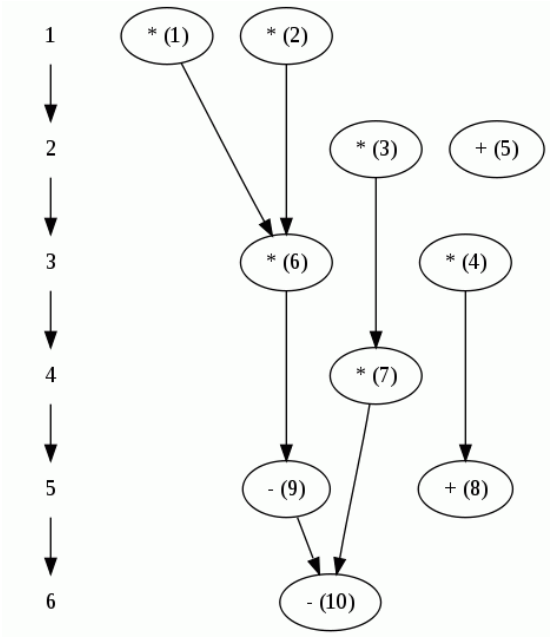
takt	Operace									
	*(1)	*(2)	*(3)	*(4)	+(5)	*(6)	*(7)	+(8)	–(9)	–(10)
1.										
2.										
3.										
4.										
5.										
6.										

Tabulka 3: Rozsah taktů, ve kterých může být operace zahájena a vyznační blokování funkční jednotky

Formulace úlohy celočíselného programování:

Funkce/nerovnice
$5 * M_{MUL} + M_{ALU}$
$x_{5,1} - M_{ALU} \leq 0$ $x_{5,2} - M_{ALU} \leq 0$ $x_{5,3} + x_{8,3} - M_{ALU} \leq 0$ $x_{5,4} + x_{8,4} - M_{ALU} \leq 0$ $x_{5,5} + x_{8,5} + x_{9,5} - M_{ALU} \leq 0$ $x_{5,6} + x_{8,6} + x_{10,6} - M_{ALU} \leq 0$
$x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} - M_{MUL} \leq 0$ $x_{1,1} + x_{2,1} + x_{3,1} + x_{3,2} + x_{4,1} + x_{4,2} - M_{MUL} \leq 0$ $x_{3,2} + x_{4,2} + x_{4,3} + x_{6,3} + x_{7,3} - M_{MUL} \leq 0$ $x_{4,3} + x_{4,4} + x_{6,3} + x_{7,3} + x_{7,4} - M_{MUL} \leq 0$ $x_{4,4} + x_{7,4} - M_{MUL} \leq 0$
Omezení odpovídající bodu 3 zůstavají nezměněna. Více viz příklad v kapitole 10.5.1.
$1 * x_{3,1} + 2 * x_{3,2} - 3 * x_{7,3} - 4 * x_{7,4} \leq -2$ $1 * x_{4,1} + 2 * x_{4,2} + 3 * x_{4,3} + 4 * x_{4,4} - 3 * x_{8,3} - 4 * x_{8,4} - 5 * x_{8,5} - 6 * x_{8,6} \leq -2$ $3 * x_{7,3} + 4 * x_{7,4} - 6 * x_{10,6} \leq -2$

Nalezené optimální řešení je $x_{1,1} = x_{2,1} = x_{3,2} = x_{4,3} = x_{5,2} = x_{6,3} = x_{7,4} = x_{8,5} = x_{9,5} = x_{10,6} = 1$, ostatní $x_{i,j} = 0$. Počet jednotek ALU = 2, násobiček = 3. Obrázek 21 znázorňuje získané řešení ve formě grafu.



Obrázek 21: Optimální plánování vícetaktových operací metodou celočíselného lineárního programování

10.5.3 Závěr

Metodou celočíselného lineárního plánování lze dosáhnout optimálního řešení. Nevýhodou je to, že s rostoucím počtem operací a rozsahem taktů, v nichž může být operace zahájena, roste i rozměr matice a počet nerovnic popisujících omezení. S rostoucím prostorem možných řešení roste časová složitost řešení soustavy nerovnic s ohledem na minimální hodnotu cenové funkce. Pro celočíselné lineární programování je dokázáno, že složitost řešení některých úloh je NP-úplná [17].

Cílů optimalizace plánování je mnoho. V této kapitole bylo plánování užito k minimalizaci ceny resp. počtu funkčních jednotek. Mezi další cíle optimalizace patří například energetická efektivita. Celočíselné lineární programování může být výhodné v případě, že má být plánováno pro více cílů zároveň. Stačí upravit funkci ceny řešení a doplnit množinu omezujících nerovnic. Platí samozřejmě, že funkce a nerovnice mohou být pouze lineární.

10.6 Iterativní heuristické plánování

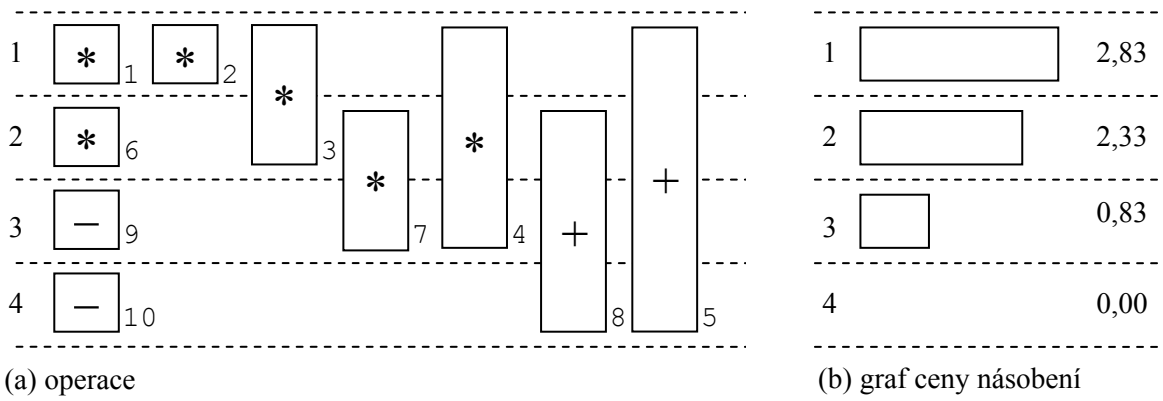
Plánování funkčních jednotek v obvodech s velkým počtem operací může metodou celočíselného lineárního programování představovat problém s časovou složitostí. Heuristické metody plánování jsou navrženy k tomu, aby nalezly vhodné řešení v polynomiálním čase. Nalezené řešení se obvykle k optimálnímu řešení pouze blíží. Metody se liší ve zvolené heuristice.

Příkladem heuristického algoritmu je iterativní algoritmus *Force-directed scheduling* (FDS)[4]. Cílem algoritmu FDS je redukce celkového počtu použitých funkčních jednotek. Algoritmus dosahuje tohoto cíle tak, že rovnoměrně rozloží operace stejného typu do výpočetních kroků (taktů). Funkční jednotky použité v jednom kroku potom mohou být opětovně využity v kroku následujícím.

Algoritmus zpočátku předpokládá, že každá operace o_i může být naplánována do každého taktu $j \in < E_i, L_i >$ se stejnou pravděpodobností. $p_j(o_i) = \frac{1}{L_i - E_i + 1}$ je pravděpodobnost, že operace o_i je naplánována do taktu j. E_i a L_i jsou hodnoty získané plánováním ASAP a ALAP. Může-li například operace o_3 být zahájena v taktu 1 nebo 2, pak $p_1(o_3) = p_2(o_3) = 0,5$. Obrázek 22 (a) znázorňuje operace algoritmu HAL (viz kapitola 10.3.1). Operace o_1, o_2, o_6, o_9, o_{10} mohou být zahájeny pouze v jediném taktu. Proto jejich pravděpodobnost $p_j(o_i)=1$. Pravděpodobnosti ostatních operací jsou $p_2(o_7) = p_3(o_7)=0,5$, $p_1(o_4) = p_2(o_4) = p_3(o_4)=0,33$, $p_2(o_8) = p_3(o_8) = p_4(o_8)=0,33$, $p_1(o_5) = p_2(o_5) = p_3(o_5) = p_4(o_5)=0,2$.

Algoritmus následně zkonstruuje grafy cen operací v jednotlivých taktech. Jeden graf se týká jednoho typu funkční jednotky (např. násobičky nebo ALU). Graf obsahuje tolik sloupců, kolik taktů potřebuje plánovaný algoritmus k dokončení. Sloupec grafu pro nějaký takt j má hodnotu rovnu součtu pravděpodobností p_j přes všechny operace, které mohou být v daném taktu zahájeny a jsou realizovány na dané funkční jednotce. Na obrázku 22 (b) je zobrazen graf ceny násobení. Hodnota sloupce 1 je dána pravděpodobností, že operace o_1, o_2, o_3, o_4 budou zahájeny v taktu 1 a tedy pro každou operaci bude potřeba vlastní násobička. Cena $2,83 = p_1(o_1)+p_1(o_2)+p_1(o_4) = 1+1+0,5+0,33$ koresponduje s počtem násobiček potřebných k provedení uvedených operací v 1. taktu. Cíl algoritmu

– redukce celkového počtu použitých funkčních jednotek – je převeden na minimalizaci maximální hodnoty sloupce grafu ceny funkční jednotky.



(a) operace (b) graf ceny násobení

Obrázek 22: Příklad algoritmu FDS. Převzato z [4] a upraveno.

Graf ceny funkční jednotky zachycuje očekávanou cenu operátoru (*expected operator cost, EOC*). Funkce velikosti *j*. sloupce grafu ceny funkční jednotky *k* je dán vzorcem $EOC_{j,k} = c_k * \sum_{i,s_j \in <E_i,L_i>} p_j(o_i)$, kde *j* je takt, *k* je typ funkční jednotky a c_k je cena funkční jednotky.

10.6.1 Algoritmus FDS

FDS je iterativní algoritmus. V každé iteraci určí takt zahájení některé operace o_i. Operaci o_i vybere z operací, které doposud nemají určený takt zahájení (tj. nejsou naplánovány). Vybraná operace je taková, že z ostatních nenaplánovaných operací minimalizuje cenu nejvíce. Cenou je tentokrát myšlena cena celého systému S: $COST(S) = \sum_{1 \le k \le m} \max_{1 \le j \le \min T} EOC_{j,k}$, kde *k* je typ funkční jednotky, *j* je takt, minT je minimální možný počet taktů potřebných k vykonání plánovaného algoritmu (viz ASAP). *COST(S)* je suma největších sloupců z grafů cen všech funkčních jednotek.

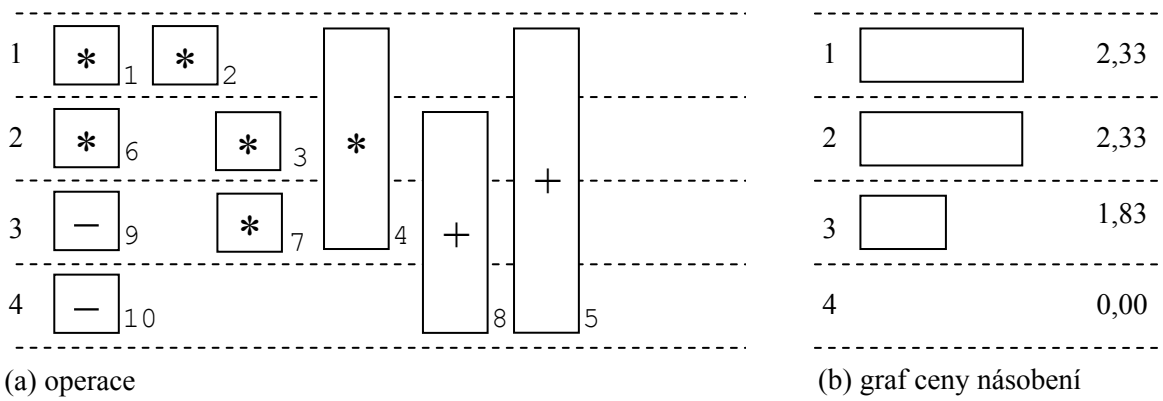
Algoritmus v pseudokódu[4]:
Vstup: *G=(U,H)*
Výstup: Plán operací $S_{current}$

```
Call ASAP(G);
Call ALAP(G);
while exists oi: Ei ≠ Li do
   MaxGain = -INFINITY;

   /* Hledání nejlepší operace k naplánování */
   for each oi: Ei ≠ Li do
      for each j: Ei ≤ j ≤ Li do
         Swork = SCHEDULE_OP(Scurrent, oi, j);
         ADJUST_DISTRIBUTION(Swork, oi, j);
         if COST(Scurrent) - COST(Swork) > MaxGain then
            MaxGain = COST(Scurrent) - COST(Swork);
```

```
                BestOp = o_i;
                BestStep = j;
            endif
        endfor
    endfor

    /* Nejlepší operace je naplánována */
    S_current = SCHEDULE_OP(S_current, BestOp, BestStep);
    ADJUST_DISTRIBUTION(S_current, BestOp, BestStep);
endwhile
```

Algoritmus ve *for* cyklu hledá nejlepší operaci k naplánování. Prochází všechny nenaplánované operace a jejich takty, v nichž mohou být zahájeny. V pomocné proměnné `S_work` udržuje kopii systému, ve kterém je operace o_i naplánována do taktu j. V proměnné `S_current` udržuje částečně naplánovaný systém po volbě nejlepší operace. Funkce `SCHEDULE_OP` naplánuje operaci o_i do taktu j a vrátí kopii systému. Funkce `ADJUST_DISTRIBUTION` projde systém resp. odpovídající graf a upraví v něm pravděpodobnosti datově závislých operací, které byly ovlivněny naplánováním operace o_i do taktu j. V proměnné `MaxGain` je udržován přínos nejlepší operace, což je největší snížení ceny systému z nenaplánovaných operací. Proměnná `BestOp` obsahuje tuto operaci a `BestStep` obsahuje takt do nějž má být operace naplánována.

Na obrázku 23 je ilustrováno naplánování zahájení operace o_3 do taktu 2. Pravděpodobnost operace o_7 musela být upravena, protože operace o_7 datově závisí na operaci o_3.



(a) operace (b) graf ceny násobení

Obrázek 23: Algoritmus FDS po naplánování operace o_3. Převzato z [4] a upraveno.

10.7 Plánování omezené prostorem

Doposud byly prezentovány algoritmy plánování omezené časem. Byl dán maximální čas řešení a algoritmus plánování měl zvolit nejmenší počet funkčních jednotek s ohledem na jejich cenu. Plánování omezené prostorem *(resource-constrained scheduling)* naproti tomu postupuje opačným směrem. Je dán maximální počet funkčních jednotek a plánovací algoritmus má najít časově nejúspornější plán výpočtu. V praxi může být dáno omezení například na velikost plochy čipu.

Algoritmy plánování omezené prostorem jsou obvykle iterativní. V každé iteraci naplánují zahájení jedné operace do nejnižšího možného taktu tak, aby byla dodržena všechna prostorová omezení. Algoritmy plánování musejí samozřejmě dodržet relaci datové závislosti danou grafem datových závislostí. Plánování operací probíhá v pořadí relace datové závislosti. Všechny datově předcházející operace musejí být naplánovány předtím, než dojde k naplánování datově závislé operace. Algortimy plánování se někdy musejí rozhodovat, kterou z připravených operací mají naplánovat dřív. Volba pořadí operací k plánování může významně ovlivnit časovou výkonnost. Existují dva známé algoritmy plánování, který tento problém řeší – plánování založené na seznamech *(list-based scheduling)* a plánování založené na statickém pořadí *(static-list scheduling)*.

10.7.1 Plánování založené na seznamech

Plánování založené na seznamech *(list-based scheduling)* je zobecněním algoritmu ASAP[4]. Algoritmus udržuje seznamy operací připravených k naplánování. Pro každý typ funkční jednotky existuje jeden seznam. Operace v seznamech jsou setříděny podle prioritní funkce.

Algoritmus postupuje tak, že vyjme ze seznamu připravených operací daného typu (např. násobení) tolik operací, kolik je k dispozici odpovídajících funkčních jednotek (např. násobiček). Tyto vyjmuté operace naplánuje všechny do jednoho taktu. Takto postupuje pro všechny seznamy. Operace jsou ze seznamu vybírány podle priority – od nejvyšší po nejnižší. Prioritní funkce tedy řeší soupeření stejných operací o funkční jednotky[4]. Naplánování operací s nižší prioritou může být odloženo do dalšího taktu. V okamžiku, kdy je některá operace naplánována, tak následně dojde k prohledání datově závislých operací. Pokud má některá datově závislá operace naplánovány všechny operace, které jí datově předcházejí, pak je vložena do seznamu daného jejím typem funkční jednotky.

Algoritmus v pseudokódu [4]:
Vstup: $G=(U,H)$,
Výstup: Plán operací $S_{current}$

```
INSERT_READY_OPS(U, PList_t1, PList_t2,… PList_tm);
int Cstep = 0;
while((PList_t1 ≠ Ø) or … or (PList_tm ≠ Ø)) do
   Cstep = Cstep + 1;
   for k=1 to m do
      for funit=1 to N_k do
         if PList_tk ≠ Ø then
            SCHEDULE_OP(S_current, FIRST(PList_tk), Cstep);
            PList_tk = DELETE(PList_tk, FIRST(PList_tk));
         ENDIF
      endfor
   endfor
   INSERT_READY_OPS(U, PList_t1, PList_t2,… PList_tm);
endwhile
```

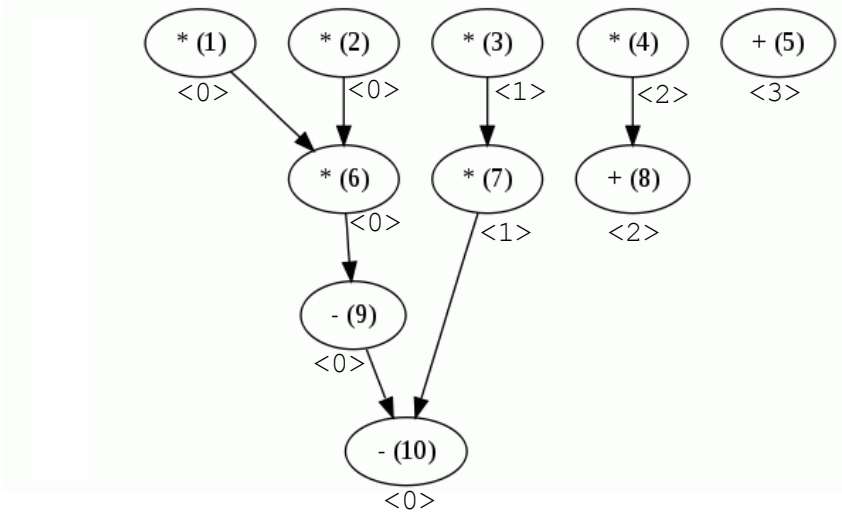
Proměnné $PList_{tk}$ jsou seznamy připravených operací daného typu t_k, $1 \leq k \leq m$, kde m je počet typů funkčních jednotek. Proměnná `Cstep` slouží jako čítač aktuálního taktu *(current step)*. Funkce `INSERT_READY_OPS` vloží do seznamů takové operace, jejichž datově předcházející operace jsou již naplánovány. Na začátku jsou do seznamů vloženy operace bez datových závislostí. Algoritmus

v cyklu `while` plánuje připravené operace, dokud nějaké v seznamech existují. Funkce `FIRST(PList`$_{tk}$`)` vrátí první připravenou operaci typu t_k s největší prioritou. Funkce `SCHEDULE_OP` naplánuje připravenou operaci do plánu $S_{current}$ do aktuálního taktu `Cstep`. Naplánovaná operace je vyjmuta ze seznamu připravených operací funkcí `DELETE`.

Významnou roli při plánování má funkce priority připravených operací `FIRST`. Některé prakticky používané funkce priority jsou:

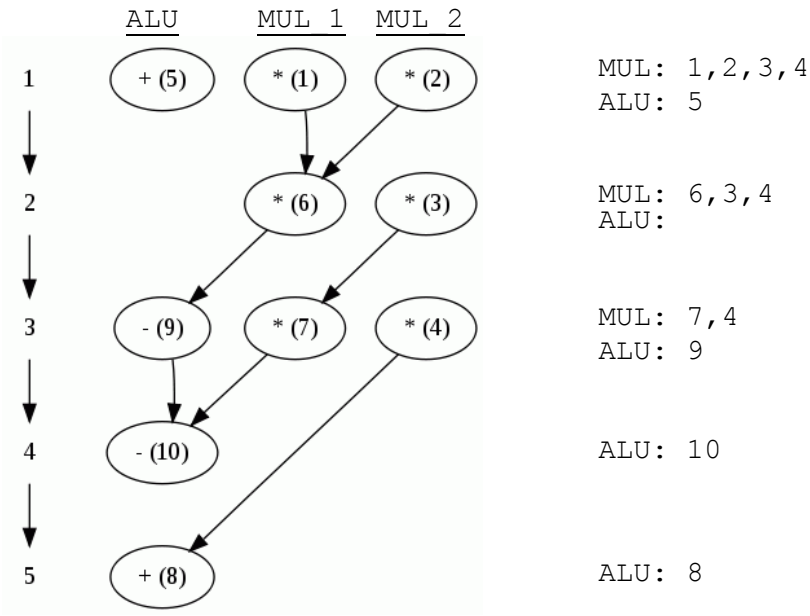
- Mobilita operace. Pro každou operaci o_i se vypočítá E_i a L_i algoritmem ASAP a ALAP. Mobilita $mrange(o_i) = L_i - E_i$ je počet taktů, v nichž může být operace o_i zahájena. Čím je mobilita operace nižší, tím větší prioritu operace má.
- Délka nejdelší cesty od operace bez datových závislostí po operaci o_i. Čím větší délka, tím vyšší priorita.
- Počet přímých datově závislých následovníků operace o_i. Priorita je uměrná počtu následovníků.

Obrázek 24 znázorňuje graf operací algoritmu HAL s vyznačenou mobilitou operací podle funkce $mrange(o_i)$. Mobilita operace je uvedena v ostrých závorkách pod elipsami. Výpočet $mrange(o_i)$ lze získat z tabulky 1 v kapitole 10.4.1.



Obrázek 24: Algoritmus HAL s vypočtenou mobilitou operací podle funkce *mrange(o$_i$)*.

Na obrázku 25 je zobrazen průběh plánování algoritmu HAL. Jsou dána omezení na počet ALU (1) a počet násobiček (2). Graf je rozdělen na pomyslné sloupce, které představují operace naplánované dané funkční jednotce. Na pravé straně se nacházejí seznamy připravených operací. Operace ze začátku seznamů jsou naplánovány odpovídajícím dostupným funkčním jednotkám.

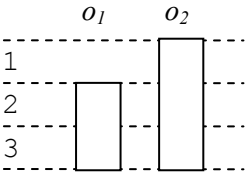


Obrázek 25: Výsledek plánování algoritmem založeným na seznamech.

10.7.2 Plánování založené na statickém pořadí

Plánování založeného na statickém pořadí se od plánování založeného na seznamech liší v tom, že místo několika seznamů tříděných dynamicky podle prioritní funkce, existuje jediný seznam operací vytvořený na začátku procesu plánování [4]. Operace jsou ze seznamu odebírány v pořadí od první po poslední položku – tak jak byl seznam sestaven. Operace odebraná z vrcholu seznamu je přiřazena volné funkční jednotce v nejnižším možném taktu s ohledem na datové závislosti mezi operacemi.

Pořadí v seznamu respektuje datové závislosti operací. Počet uspořádání seznamu s ohledem na datové závislosti může být velký. Na výběru vhodné strategie uspořádání závisí efektivita výsledného řešení. Autoři [4] navrhují uspořádání seznamu tak, že operace jsou primárně setříděny podle atributu L_i (ALAP) vzestupně a sekundárně podle atributu E_i (ASAP) sestupně. Připomeňme, že dvě operace o_i a o_j, pro které platí $o_i \rightarrow^+ o_j$ mají $L_i < L_j$, kde $\rightarrow^+$ je tranzitivní uzávěr relace přímé datové závislosti. Primární setřídění podle atributu L_i by již vedlo ke korektnosti výsledného řešení. Sekundární setřídění podle atributu E_i (sestupně) dává při plánování přednost operaci s vyšší hodnotou E_i. Obrázek 26 znázorňuje dvě operace, které mají shodný atribut L_i, a proto budou dále uspořádány podle atributu E_i: o_1 první, o_2 druhá. Přednost tedy dostává operace s nižší mobilitou (viz kapitola 10.7.1).



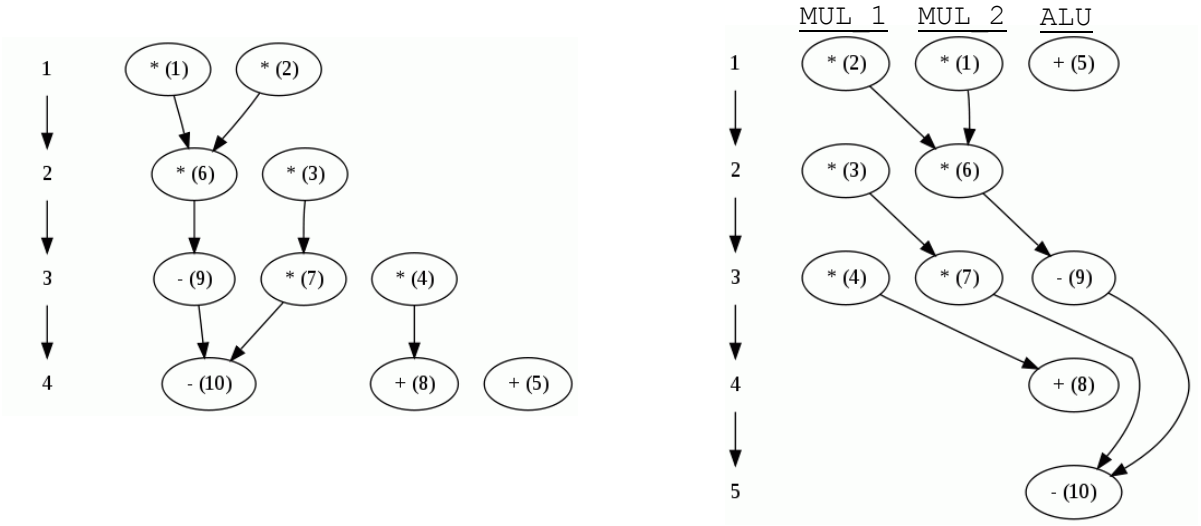
	E_i	L_i
o_1	2	3
o_2	1	3

Obrázek 26: Uspořádání operací

Tabulka 4 a obrázek 27 ilustrují plánování založené na statickém pořadí. Pořadí je počítáno podle algoritmů ALAP a ASAP. Počet násobiček je omezen na dvě, ALU je jedna.

operace	–(10)	+(8)	+(5)	–(9)	*(7)	*(4)	*(6)	*(3)	*(1)	*(2)
ALAP	4	4	4	3	3	3	2	2	1	1
ASAP	4	2	1	3	2	1	2	1	1	1
pořadí	10.	9.	8.	7.	6.	5.	4.	3.	2.	1.

Tabulka 4: výpočet statického pořadí operací pro příklad algoritmu HAL



(a) plánování ALAP (b) plánování založené na statickém pořadí

Obrázek 27: příklad plánování založeného na statickém pořadí

10.7.3 Plánování založené na celočíselném lineárním programování

Problém plánování s prostorovými omezeními lze formulovat jako úlohu celočíselného lineárního programování. Autor této práce se domnívá, že toto zatím nebylo nikým publikováno. Výstupem plánování je optimální časový plán, tedy nikoliv heuristické řešení jako v případě plánování založeného na seznamech nebo statickém pořadí.

Formulace úlohy lineárního programování podléhá několika předpokladům:

1. Je potřeba určit maximální počet taktů potřebných k dokončení výpočtu *(maxT)*. Maximální počet taktů bude potřeba tehdy, když budou všechny operace vykonávány sekvenčně.
2. Operace o_i může být naplánována v rozsahu taktů $<E_i, L_i>$, kde L_i je ALAP se vstupem rovným maximálnímu počtu taktů z předchozího bodu.
3. Počet funkčních jednotek je předem dán. Algoritmus plánování jej nehledá.
4. Funkce ceny řešení minimalizuje maximální číslo taktu, v němž je naplánována nějaká operace.

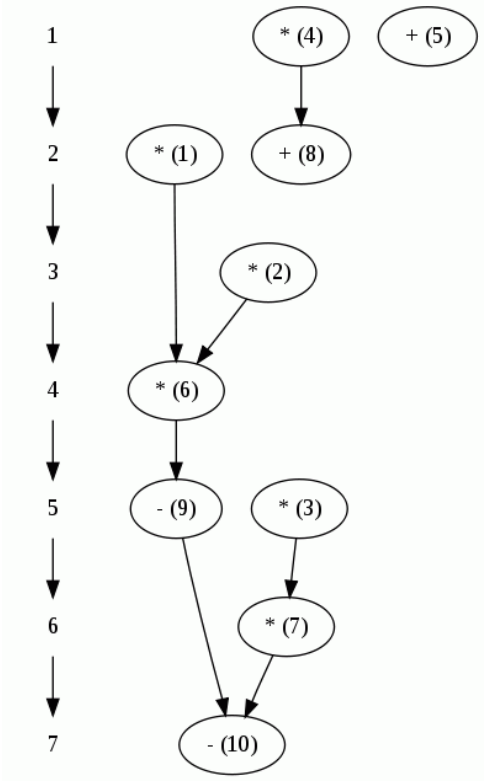
Úloha celočíselného lineárního programování může být formulována takto:

1. Minimalizuj hodnotu cenové funkce: $\sum_{i=1}^{n} \sum_{j=1}^{\max T} j * x_{i,j}$. Stejně jako v kapitole 10.5.1 je operace o_i naplánována do taktu j, pokud $x_{i,j}=1$. Proměnná j slouží jako váha taktu. V lineárním programu se projeví jako konstanta, tedy úloha zůstává stále lineární. Smysl váhy taktu je

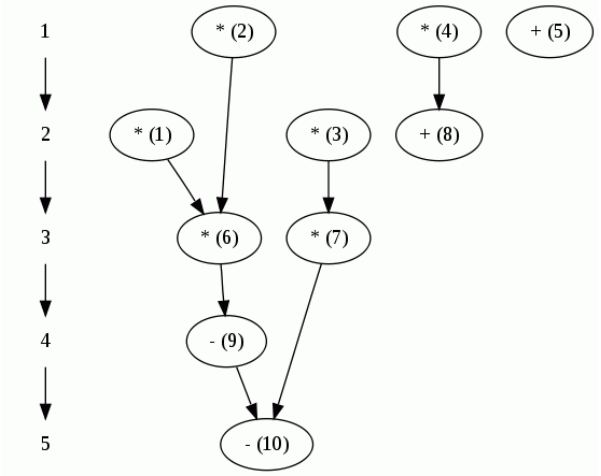
následující: jestliže nějaká operace o_i může být naplánována do dvou (nebo více) taktů, tak díky váze taktů bude zvolen nižší takt, aby hodnota cenové funkce byla minimální.

2. Omezení $\sum_{\substack{i=1 \\ o_i \in FU_{th}}}^{n} x_{i,j} - M_{tk} \leq 0,$ pro $1 \leq j \leq \min_T, 1 \leq k \leq m$ říká, že v každém taktu výpočtu (j) má být použito maximálně M_{tk} funkčních jednotek typu t_k. Na rozdíl od kapitoly 10.5.1 jsou M_{tk} konstanty.
3. Omezení $\sum_{j=1}^{\max T} x_{i,j} = 1$ pro $1 \leq i \leq n$ říká, že každá operace o_i smí být zahájena pouze jednou.
4. Vazby mezi operacemi jsou dodrženy díky omezení $\sum_{j=1}^{\max T} j * x_{i,j} - \sum_{j=1}^{\max T} j * x_{k,j} \leq -1$.

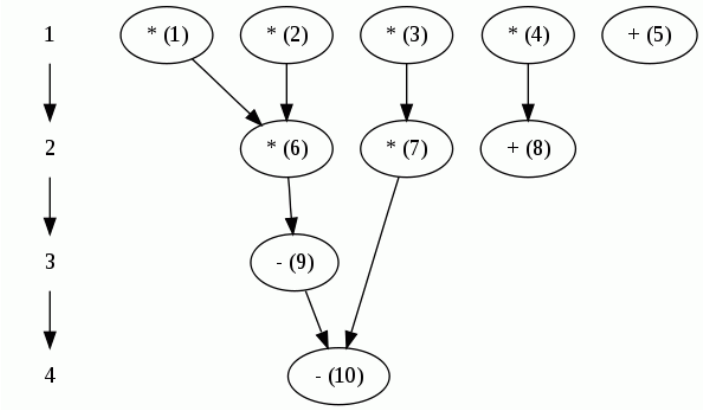
Plánování je demonstrováno na příkladu algoritmu HAL. Obrázky 28 a 30 znázorňují krajní případy omezení pro minimální a neomezený počet funkčních jednotek. Plán pro neomezený počet funkčních jednotek je ekvivalentní plánu ASAP. Obrázek 29 znázorňuje řešení pro 1 ALU a 2 násobičky.



Obrázek 28: plán pro 1 ALU a 1 násobičku



Obrázek 29: plán pro 1 ALU a 2 násobičky



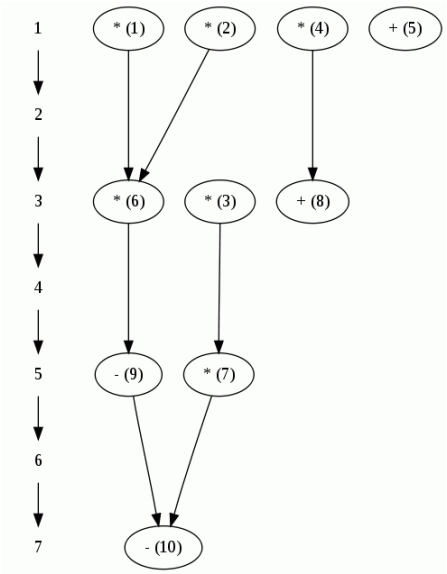
Obrázek 30: plán pro neomezený počet funkčních jednotek

10.7.3.1 Vícetaktové operace

Rozšíření úlohy celočíselného lineárního programování o vícetaktové operace je snadné. Stačí změnit omezení 2 a 4, podobně jako v kapitole 10.5.2.

- (omezení 2'): $\sum_{\substack{i=1 \\ o_i \in FU_{tk}}}^{n} \sum_{p=0}^{latence_i - 1} x_{i,j-p} - M_{tk} \leq 0$ pro $1 \leq j \leq \min_T, 1 \leq k \leq m$.
- (omezení 4'): $\sum_{j=1}^{\max T} j * x_{i,j} - \sum_{j=1}^{\max T} j * x_{k,j} \leq -latence_i$ pro všechny relace precedence mezi operacemi $o_i \rightarrow o_k$.

Na obrázku 31 je zachyceno plánování systému HAL pro 1 ALU a 3 násobičky. Latence násobení jsou 2 takty, sčítání a odčítání 1 takt.



Obrázek 31: plánování vícetaktových operací s omezením na 1 ALU a 3 násobičky

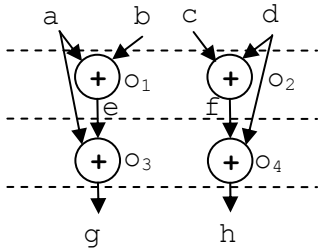
11 Alokace zdrojů

Alokace zdrojů je dalším krokem v návrhu procesoru. Fáze alokace zdrojů mapuje funkční jednotky na operace, paměťové prvky na proměnné. Mezi funkčními jednotkami a paměťovými prvky vedou signály a sběrnice. Algoritmy alokace se snaží minimalizovat počet funkčních jednotek, paměťových prvků a hustotu propojení pomocí sdílení signálů a sběrnic.

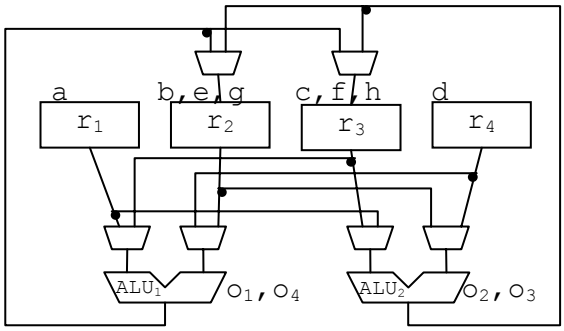
Obrázek 32 znázorňuje výsledek plánování skupiny příkazů a alokaci zdrojů. Doby života opoměnných `b,e,g` a `c,f,h` se nepřekrývají. Proto mohou sdílet společný registr. Na obrázku (c) je zobrazena varianta alokace zdrojů, která optimalizuje návrh na počet použitých registrů a funkčních jednotek. V důsledku toho je nutné přidat multiplexory do datových cest. Lepší varianta alokace je uvedena na obrázku (d). Kromě optimalizace počtu registrů a funkčních jednotek je v úvahu brána i složitost datových cest. Tím došlo k ušetření multiplexorů.

```
e=a+b;
g=a+e;
f=c+d;
h=f+d;
```

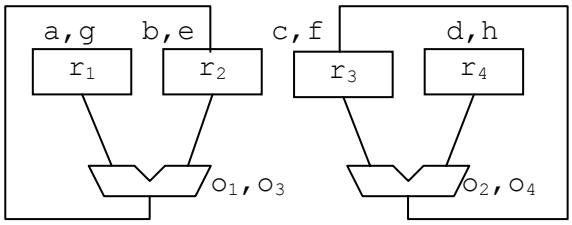
(a) posloupnost příkazů



(b) výsledek plánování operací



(c) neoptimální alokace zdrojů



(d) optimální alokace zdrojů

Obrázek 32: Alokace zdrojů. Převzato z [4].

Proces alokace lze rozdělit do několika kroků [4]:

- výběr jednotek *(unit selection)*,
- mapování jednotek *(unit binding)*,
- mapování paměťových prvků *(storage binding)*,
- konstrukce propojovacích sítí *(interconnection binding)*.

Některé operace lze vykonávat na více typech funkčních jednotek. Například sčítání lze provézt na sčítačce nebo na ALU společně s odčítáním a porovnáváním. Ve fázi výběru jednotek by mělo dojít k rozhodnutí, jaká množina jednotek se bude následně mapovat na operace a proměnné. Obecně platí, že počet funkčních jednotek daného typu musí pokrýt všechny operace stejného typu plánované do jednoho kroku. V příkladu na obrázku 32 byly zvoleny dvě ALU, neboť operace sčítání probíhají v obou taktech paralelně. Výběr jednotek bývá obvykle kombinován s mapováním jednotek.

V každém taktu výpočtu musejí být všechny operace namapovány na funkční jednotky. Bývá výhodné sdílet funkční jednotky mezi operacemi, které nemohou běžet paralelně. Algoritmus mapování by měl namapovat sdílené funkční jednotky na operace tak, aby došlo k minimalizaci propojení a zjednodušení datových cest. Z obrázku 32 je patrné, že nevhodným sdílením funkčních jednotek (ALU) lze dosáhnout zhoršení propojení a přídání multiplexorů.

Paměťové prvky jsou registry, paměti RAM a ROM. Proměnné mohou často sdílet jeden registr nebo buňky paměti, pokud se jejich doba životnosti nepřekrývá. Za účelem minimalizace propojení je výhodné seskupovat registry do registrových polí. Počet registrů v registrovém poli použitelných v jednom taktu závisí na počtu vstupních a výstupních portů registrového pole.

Propojovací sítě spojují paměťové prvky a funkční jednotky. Opět bývá výhodné sdílení propojovací cesty, pokud není používána více jednotkami v jednom taktu. Na obrázku 32 (c) jsou sdílené propojovací cesty od ALU k registrům.

Kroky procesu alokace se navzájem ovlivňují. Záleží na pořadí, v jakém budou vykonány. Algoritmy alokace lze rozdělit na konstruktivní *(greedy constructive)*, dekompoziční a iterativní *(iterative refinement)*.

11.1 Konstruktivní algoritmus

Konstruktivní algoritmus [4] používá hladový přístup pro mapování jednotek a paměťových prvků na operace a proměnné. Algoritmus v každé iteraci namapuje jedno zařízení. Zvolí takové, které zvýší cenu řešení nejméně. Algoritmus se snaží o znovupoužití funkčních jednotek, pokud nebyly pro daný takt namapovány pro jinou operaci. Jsou-li dostupné dvě a více funkčních jednotek pro vykonání dané operace, pak se vybere taková, která minimalizuje cenu propojení. Není-li dostupná žádná funkční jednotka, pak algoritmus vytvoří novou instanci funkční jednotky. Podobně se algoritmus zachová při mapování registrů na proměnné, přičemž bere v úvahu životnost proměnné. Mezi mapované zdroje patří také signály. Signály lze mapovat na datové toky (např. mezi proměnnou a operací). Cílem je využít jeden signál pro více datových toků (např. od registru k funkční jednotce). Datové toky mohou sdílet signál, pokud v nich nedochází k přenosu dat ve stejném taktu.

Algoritmus v pseudokódu:
Vstup: množina entit `UBE` k namapování na fyzická zdroje
Výstup: datová cesta $DP_{current}$ s entitami namapovanými na zařízení

```
DP_current = ∅;
while (UBE ≠ ∅) do
   LowestCost = INFINITY;

   for all ube∈UBE do
      DP_work = ADD(DP_current, ube);
      c_work = COST(DP_work);
      if (c_work < LowestCost) then
         LowestCost = c_work;
         BestEntity = ube;
      endif
   endfor

   DP_current = ADD(DP_current, BestEntity);
   UBE = UBE – BestEntity;
endwhile
```

Algoritmus dostane na vstupu množinu entit (operací, proměnných, datových toků) `UBE`, které mají být namapovány na zdroje (funkční jednotky, registry, signály). V každé iteraci *while* cyklu projde všechny nenamapované entity a zjistí, která z nich může být přidána do výsledné datové cesty $DP_{current}$, aby celkovou cenu řešení zvýšila minimálně. Funkce ADD přidá entitu do datové cesty, přičemž znovupoužije některé volné zdroje nebo vytvoří novou instanci zdroje (viz výše). Funkce ADD vrátí novou datovou cestu. Funkce COST vypočítá cenu datové cesty. Na konci iterace je v proměnné `BestEntity` nejvhodnější entita k namapování. Ta je následně přidána do datové cesty a odebrána z dalšího zpracování.

11.2 Rozdělení na kliky

Alokace rozdělováním na kliky *(clique partitioning)*[4] je příkladem dekompozičního přístupu. Algoritmy s dekompozičním přístupem provedou nejprve alokaci jednoho typu zdroje, následně dalšího atd. Například mohou začít s mapováním registrů na proměnné, následně funkčních jednotek na operace a nakonec mapování signálů na datové toky. Díky tomu, že jednotlivé kroky se navzájem ovlivňují, tak výsledné řešení nemusí být optimální, i kdyby byl každý krok vyřešen optimálně.

Algoritmus rozdělení na kliky se snaží namapovat určitý zdroj na co nejvíc entit. Algoritmus sestaví pro daný typ entity (např. proměnné) neorientovaný graf. Uzly jsou entity. Hrana spojuje dvě entity, které mohou sdílet společný zdroj (např. registr). Množina entit může sdílet jeden zdroj, pokud jsou v grafu hustě propojeny, tedy tvoří úplný podgraf – kliku. Zdroje jsou nejhospodárněji využity tehdy, když je celý graf rozdělen na úplné podgrafy maximální velikosti, tedy počet úplných podgrafů je minimální. Každému úplnému podgrafu odpovídá jeden zdroj.

Problém rozdělení grafu na kliky je NP-úplný [15]. Z tohoto důvodu je obvykle řešen pomocí heuristiky.

11.2.1 Alokace registrů

Pro alokaci registrů je sestaven graf, kde uzly jsou proměnné a hrany spojují takové proměnné, jejichž doby životnosti se nepřekrývají (tedy mohou sdílet společný registr).

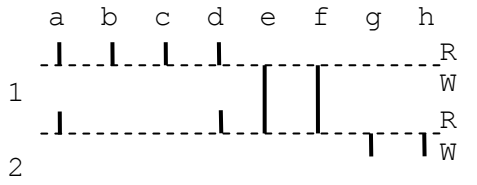
Příklad: nechť je dána posloupnost příkazů z obrázku 32. Obrázek 33 znázorňuje doby životnosti jednotlivých proměnných. Vlevo jsou čísla taktů. Každý takt lze z pohledu registru rozdělit na dva časové úseky – čtení a zápis. Díky konstrukci registru jako D-klopného obvodu lze z registru číst a do něj zároveň zapisovat v jednom taktu, aniž by došlo ke kolizi. Na obrázku 34 je zobrazen graf proměnných. Jedno z možných rozdělení grafu proměnných na kliky je uvedeno na obrázku 35. Proměnné by byly mapovány do čtyř registrů podle klik na obrázku:

- registr r_1 pro proměnnou *a*,
- registr r_2 pro proměnné *b*, *e*, *g*,
- registr r_3 pro proměnnou *c*, *f*, *h*,
- registr r_4 pro proměnnou *d*.

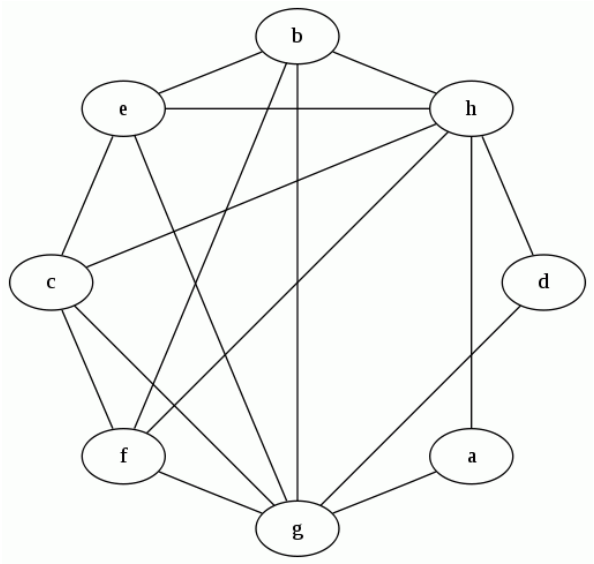
Řešení bude vždy obsahovat dvě kliky o třech uzlech (proměnných) a dvě kliky o jednom uzlu.

Algoritmus musí zvolit jedno mapování registrů na proměnné. S tímto mapováním se bude pracovat v dalších krocích alokace jako s pevně daným.

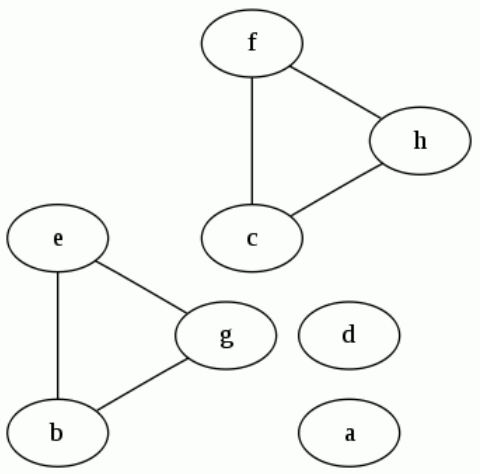
```
e=a+b;
g=a+e;
f=c+d;
h=f+d;
```



Obrázek 33: doby životnosti proměnných



Obrázek 34: graf proměnných



Obrázek 35: rozdělení grafu proměnných na kliky

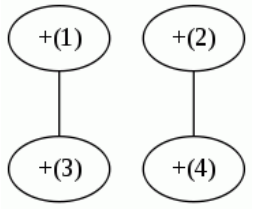
11.2.2 Alokace funkčních jednotek

Graf pro alokaci funkčních jednotek je sestaven takto[4]:

- uzly jsou operace,
- hrana mezi uzly existuje tehdy, když
    a) operace v uzlech jsou naplánovány do různých taktů (tj. běhy se nepřekrývají),
    b) existuje funkční jednotka, která dokáže vykonat obě operace.

Klika v grafu odpovídá mapování operací na funkční jednotku.

Příkladu z obrázku 32 odpovídá graf operací na obrázku 36. K výpočtu budou použity dvě ALU: ALU_1 pro o_1,o_3 a ALU_2 pro o_2,o_4.

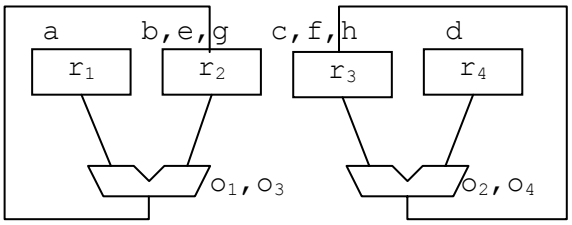


Obrázek 36: graf operací

11.2.3 Alokace signálů

Datové toky (např. mezi registry a funkčními jednotkami) mohou sdílet společné signály. Vrcholy grafu jsou datové toky (např. (r_1, ALU_1) pro tok dat z registru r_1 do funkční jednotky ALU_1). Hrana existuje mezi datovými toky, které neprobíhají souběžně. Klika v grafu představuje signál, který je sdílený pro datové toky v uzlech kliky.

V příkladu z obrázku 32 neexistuje žádná hrana, tudíž signály nebudou sdíleny. Řešení získané alokací registrů, funkčních jednotek a signálů rozdělením na kliky je zobrazeno na obrázku 37.



Obrázek 37: alokace rozdělením na kliky

11.3 Algoritmus iterativního vylepšování

Algoritmus iterativního vylepšování[4] *(iterative refinement approach)* se snaží vylepšit alokaci zdrojů, kterou obdržel na vstupu od jiného algoritmu. Konkrétní algoritmy iterativního vylepšování se liší v tom, jakou metodu vylešování volí. Jako jedna z výhodných strategií se jeví realokace zdrojů. Algoritmus vezme dvě entity (např. dvě proměnné) na dvou různých zdrojích (např. v registrech) a pokusí se je navzájem vyměnit. Algoritmus spočítá, zda získané řešení je výhodnější než aktuální řešení. Příkladem úspěšného vylepšení je výměna operace o_4 (v ALU_1) za o_3 (v ALU_2) na obrázku 32. Tím bylo získáno výhodnější řešení alokace, jemuž odpovídá obrázek 37.

Algoritmus iterativního vylepšování alokace operací[4]:
Vstup/Výstup: datová cesta $DP_{current}$ s entitami namapovanými na zařízení

```
repeat
   BestGain = -INFINITY;
   c_current = COST(DP_current);
   for each s∈control steps do
      for each o_i,o_j, type(o_i)==type(o_j), i≠j, scheduled into s do
         DP_work = SWAP(DP_current, o_i, o_j);
         c_work = COST(DP_work);
         CurrentGain = c_current – c_work;
         if CurrentGain > BestGain then
            BestGain = CurrentGain;
            BestOp1 = o_i; BestOp2 = o_j;
         endif
      endfor
   endfor

   if BestGain>0 then
      DP_current = SWAP(DP_current,BestOp1,BestOp2);
   endif
until BestGain ≤ 0
```

Uvedený algoritmus využívá hladového přístupu k řešení problému realokace. Algoritmus projde všechny dvojice operací o_i, o_j, které jsou stejného typu a mají zahájení naplánováno na stejný takt. Algoritmus zkusí vyměnit obě operace mezi jejich funkčními jednotkami. Vypočítá cenu zlepšení

oproti aktuálnímu návrhu alokace. Dvojice operací, které nejvíce přispějí k vylepšení alokace jsou uloženy v proměnných `BestOp1`, `BestOp2`. Tyto dvě operace jsou na konci každé iterace prohozeny mezi svými funkčními jednotkami. Algoritmus provádí zlepšování, dokud existují nějaké dvě operace, jejichž prohození by vedlo k nějakému vylepšení.

Algoritmus má nevýhodu, že díky hladovému přístupu neprohledá všechna možná vylepšení. Algoritmus lze upravit, aby prováděl realokaci proměnných místo operací, případně obou (podle vzoru konstruktivního algoritmu v kapitole 11.1).

12 Závěr

V této práci byly popsány nejvýznamnější kroky transformace vyššího programovacího jazyka na jazyk pro popis hardwaru. Bylo ukázáno, že vhodnou vnitřní reprezentaci programu představuje kombinavý graf datových závislostí a toku řízení, protože v tomto grafu lze snadno odhalit paralelizmus výpočtů.

Významná část práce je věnována optimalizacím, které vycházejí z algebraických úprav. Algebraické úpravy lze provádět díky znalostem základních vlastností operátorů: asociativity, komutativity a distributivity. V této oblasti probíhá intenzivní výzkum. Důkazem toho je fakt, že modifikace Huffmanova algoritmu pro redukci výšky stromu výrazů byla poprvé publikována teprve v roce 2008. Autor této práce věří, že přispěl k této oblasti výzkumu zejména díky analýze optimalizace distributivních operátorů.

Prezentované metody plánování pokrývají širokou oblast postupů používaných v praxi. Zvláštní pozornost byla věnována celočíselnému lineárnímu programování, protože se jedná o snadný způsob návrhu optimalizačních algoritmů. Mnoho výzkumných prací v oblasti plánování bývá nejprve řešeno metodou celočíselného lineárního programování a až následně bývají vyvíjeny specializované (např. heuristické) algoritmy s lepší časovou složitostí. Autor této diplomové práce navrhl formulaci úlohy celočíselného lineárního programování pro plánování omezené prostorem. Autor vidí budoucnost výzkumu v oblasti plánování v optimalizaci obvodů pro nízkou spotřebu a bezpečnost. Další směr výzkumu by se měl orientovat na návrh optimálizovaných řetězených architektur.

Algoritmy pro alokaci zdrojů řeší obecně těžké problémy. Prezentované algoritmy nejsou schopné nalézt optimální řešení v obecně rozumném (tj. polynomiálním) čase. Autor spatřuje možnost k přispění do problematiky alokace ve výzkumu strategií algoritmu iterativního vylepšování.

Praktická část diplomové práce byla zaměřena na konstrukci zadní části překladače z jazyka ISAC do jazyka VHDL. Jako vedlejší produkt studia redukce výšky stromu výrazů a algoritmů plánování byl vytvořen program, který optimalizuje kód základního bloku programu a vizualizuje provedené optimalizace ve formě grafů operací.

Literatura

[1] Kolektiv autorů: P6 Family of Processors Hardware Developer's Manual [online]. Intel Corporation, 1998 [cit. 2009-12-23]. Dostupné na URL: http://download.intel.com/design/PentiumII/manuals/24400101.pdf

[2] Dvořák, V., Drábek, V.: Architektura procesorů. VUTIUM, Brno, 1999, 303 s., ISBN 80-214-1458-8

[3] Hwang E. O.: Digital Logic and Microprocessor Design with VHDL. Thomson, 2006. ISBN 0-534-46593-5

[4] Gajski, D.D., Dutt , N., Wu A., Lin S.: High-Level Synthesis. Kluwer, Boston, 1992. ISBN 07-923-9194-2

[5] Hoffmann, A., Meyr, H., Leupers, R.: Architecture Exploration for Embedded Processors with LISA. Kluwer Academic Publischers, Dordrecht, 2002. ISBN 14-020-7338-0

[6] Češka, M., Vojnar, T., Smrčka, A.: Teoretická informatika (studijní opora). VUT, 2007

[7] Aho, A. V., Sethi, R., Ullman, J. D., Compilers: Principles, Techniques, and Tools. Bell Laboratories, 1986. ISBN 0-201-10088-6

[8] Coons, K. a kol.: Optimal Huffman Tree-Height Reduction for Instruction-Level Parallelism. Dostupné na URL: http://www.cs.utexas.edu/users/mckinley/380C/lecs/tree height-tr-2008.pdf

[9] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, K. Zadeck: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems, vol. 13, pp. 451–490, Oct. 1991.

[10] Gao, J.: Analysis of Algorithms (lecture notes). Stony Brook University, 2007. Dostupné na URL: http://www.cs.sunysb.edu/~jgao/CSE548-fall07/David-mount-DP.pdf

[11] Hu, T. C., Shing, M. T.: Computation of matrix chain products. SIAM Journal on Computing (Univ. of California at San Diego: Springer-Verlag) 13 (2), ISSN 0097-5397

[12] Kolektiv autorů: Factoring. Dostupné na URL: http://library.thinkquest.org/20991/alg/factoring.html (2010)

[13] Knuth, D.: The Art of Computer Programming, Volume 2, 2nd ed. Addison-Wesley Publishing, 1981, ISBN 0-201-03822-6

[14] Češka, M., Marek, V., Novosad, P., Vojnar, T.: Petriho sítě (studijní opora). VUT, 2009

[15] Skiena, Steven S.: The algoritmh design manual. Springer Science+Business Media, LLC, New York, 1998, ISBN 0-387-94860-7

[16] Paulin, P. G., Knight, J. P.: Scheduling and binding algorithms for high-level synthesis. Proceedings of the 26th ACM/IEEE Design Automation Conference, Las Vegas, 1989, ISBN 0-89791-310-8

[17] Matoušek, J., Gartner, B.: Understanding and using linear programming. Springer-Verlag, 2007, ISBN 3-540-30697-8

[18] J.-H. Lee, Y.-C. Hsu and Y.-L. Lin, A New Integer Linear Programming Formulation for the Scheduling Problem in Data Path Synthesis, ICCAD, 1989

[19] Hruška, T., Masařík, K., Zámečníková, E.: ISAC manual (version 2), Fakulta informačních technologií VUT, Brno, 2009